Ada Reference Manual

2022 Edition

with 2025 changes

Language and Standard Libraries

Copyright © 1992, 1993, 1994, 1995 Intermetrics, Inc.

Copyright © 2000 The MITRE Corporation, Inc.

Copyright © 2004, 2005, 2006 AXE Consultants

Copyright © 2004, 2005, 2006 Ada-Europe

Copyright © 2008, 2009, 2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023, 2024, 2025 AXE Consultants

Ada Reference Manual - Language and Standard Libraries

Copyright © 1992, 1993, 1994, 1995, Intermetrics, Inc.

This copyright is assigned to the U.S. Government. All rights reserved.

This document may be copied, in whole or in part, in any form or by any means, as is or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Compiled copies of standard library units and examples need not contain this copyright notice so long as the notice is included in all copies of source code and documentation.

Technical Corrigendum 1

Copyright © 2000, The MITRE Corporation. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of MITRE.

You use this document on the condition that you indemnify and hold harmless MITRE, its Board of Trustees, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

MITRE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL MITRE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF MITRE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Amendment 1

Copyright © 2004, 2005, 2006, 2007, AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or

third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Third Edition (Ada 2012)

Copyright © 2008, 2009, 2010, 2011, 2012 AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Technical Corrigendum 1 for Ada 2012

Copyright © 2013, 2014, 2015, 2016 AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Fourth Edition (Ada 2022)

Copyright © 2016, 2017, 2018, 2019, 2020, 2021, 2022, 2023 AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Amendment 1 for Ada 2022

Copyright © 2023, 2024, 2025 AXE Consultants. All Rights Reserved.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of AXE.

You use this document on the condition that you indemnify and hold harmless AXE, its board, officers, agents, and employees, from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

AXE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL AXE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF AXE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Ada 2005 Consolidated Reference Manual

Copyright © 2004, 2005, 2006, Ada-Europe.

This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy. Any other use or distribution of this document is prohibited without the prior express permission of Ada-Europe.

You use this document on the condition that you indemnify and hold harmless Ada-Europe and its Board from any and all liability or damages to yourself or your hardware or software, or third parties, including attorneys' fees, court costs, and other related costs and expenses, arising out of your use of this document irrespective of the cause of said liability.

ADA-EUROPE MAKES THIS DOCUMENT AVAILABLE ON AN "AS IS" BASIS AND MAKES NO WARRANTY, EXPRESS OR IMPLIED, AS TO THE ACCURACY, CAPABILITY, EFFICIENCY MERCHANTABILITY, OR FUNCTIONING OF THIS DOCUMENT. IN NO EVENT WILL ADA-EUROPE BE LIABLE FOR ANY GENERAL, CONSEQUENTIAL, INDIRECT, INCIDENTAL, EXEMPLARY, OR SPECIAL DAMAGES, EVEN IF ADA-EUROPE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Table of Contents

Table of Contents i
Foreword xiii
Introduction xiv
1 General 1
1.1 Scope 1
1.1.1 Extent 1
1.1.2 Structure 1
1.1.3 Conformity of an Implementation 3
1.1.4 Method of Description and Syntax Notation 5
1.1.5 Classification of Errors 6
1.2 Normative References 7
1.2.1 Bibliography 8
1.3 Terms and Definitions 8
1.3.1 Types, Objects, and their Properties 8
1.3.2 Subprograms and their Properties 12
1.3.3 Other Syntactic Constructs 13
1.3.4 Runtime Actions 15
1.3.5 Exceptional Situations 15
2 Lexical Elements 17
2.1 Character Set 17
2.2 Lexical Elements, Separators, and Delimiters 19
2.3 Identifiers 21
2.4 Numeric Literals 22
2.4.1 Decimal Literals 22
2.4.2 Based Literals 22
2.5 Character Literals 23
2.6 String Literals 23
2.7 Comments 24
2.8 Pragmas 24
2.9 Reserved Words 27
3 Declarations and Types 29
3.1 Declarations 29
3.2 Types and Subtypes 30
3.2.1 Type Declarations 32
3.2.2 Subtype Declarations 33
3.2.3 Classification of Operations 34
3.2.4 Subtype Predicates 35
3.3 Objects and Named Numbers 38
3.3.1 Object Declarations 41
3.3.2 Number Declarations 44
3.4 Derived Types and Classes 44
3.4.1 Derivation Classes 48
3.5 Scalar Types 49
3.5.1 Enumeration Types 53
3.5.2 Character Types 54
3.5.3 Boolean Types 55

3.5.4 Integer Types 55
3.5.5 Operations of Discrete Types 58
3.5.6 Real Types 59
3.5.7 Floating Point Types 60
3.5.8 Operations of Floating Point Types 62
3.5.9 Fixed Point Types 62
3.5.10 Operations of Fixed Point Types 64
3.6 Array Types 65
3.6.1 Index Constraints and Discrete Ranges 68
3.6.2 Operations of Array Types 69
3.6.3 String Types 70
3.7 Discriminants 71
3.7.1 Discriminant Constraints 74
3.7.2 Operations of Discriminated Types 75
3.8 Record Types 75
3.8.1 Variant Parts and Discrete Choices 78
3.9 Tagged Types and Type Extensions 80
3.9.1 Type Extensions 83
3.9.2 Dispatching Operations of Tagged Types 85
3.9.3 Abstract Types and Subprograms 88
3.9.4 Interface Types 90
3.10 Access Types 92
3.10.1 Incomplete Type Declarations 95
3.10.2 Operations of Access Types 97
3.11 Declarative Parts 103
3.11.1 Completions of Declarations 104
4 Names and Expressions 105
4.1 Names 105
4.1.1 Indexed Components 106
4.1.2 Slices 107
4.1.3 Selected Components 108
4.1.4 Attributes 110
4.1.5 User-Defined References 112
4.1.6 User-Defined Indexing 113
4.2 Literals 115
4.2.1 User-Defined Literals 116
4.3 Aggregates 118
4.3.1 Record Aggregates 118
4.3.2 Extension Aggregates 121
4.3.3 Array Aggregates 122
4.3.4 Delta Aggregates 127
4.3.5 Container Aggregates 129
4.4 Expressions 135
4.5 Operators and Expression Evaluation 137
4.5.1 Logical Operators and Short-circuit Control Forms 138
4.5.2 Relational Operators and Membership Tests 139
4.5.3 Binary Adding Operators 144
4.5.4 Unary Adding Operators 145
4.5.5 Multiplying Operators 145
4.5.6 Highest Precedence Operators 147
4.5.7 Conditional Expressions 148
4.5.8 Quantified Expressions 150

4.5.9 Declare Expressions 151
4.5.10 Reduction Expressions 152
4.6 Type Conversions 155
4.7 Qualified Expressions 161
4.8 Allocators 162
4.9 Static Expressions and Static Subtypes 164
4.9.1 Statically Matching Constraints and Subtypes 167
4.10 Image Attributes 168
5 Statements 175
5.1 Simple and Compound Statements - Sequences of Statements 175
5.2 Assignment Statements 177
5.2.1 Target Name Symbols 178
5.3 If Statements 179
5.4 Case Statements 180
5.5 Loop Statements 181
5.5.1 User-Defined Iterator Types 184
5.5.2 Generalized Loop Iteration 187
5.5.3 Procedural Iterators 190
5.6 Block Statements 194
5.6.1 Parallel Block Statements 194
5.7 Exit Statements 196
5.8 Goto Statements 196
6 Subprograms 199
6.1 Subprogram Declarations 199
6.1.1 Preconditions and Postconditions 202
6.1.2 The Global and Global'Class Aspects 207
6.2 Formal Parameter Modes 210
6.3 Subprogram Bodies 211
6.3.1 Conformance Rules 212
6.3.2 Inline Expansion of Subprograms 214
6.4 Subprogram Calls 215
6.4.1 Parameter Associations 217
6.5 Return Statements 221
6.5.1 Nonreturning Subprograms 224
6.6 Overloading of Operators 225
6.7 Null Procedures 226
6.8 Expression Functions 226
7 Packages 229
7.1 Package Specifications and Declarations 229
7.2 Package Bodies 230
7.3 Private Types and Private Extensions 231
7.3.1 Private Operations 234
7.3.2 Type Invariants 236
7.3.3 Default Initial Conditions 240
7.3.4 Stable Properties of a Type 241
7.4 Deferred Constants 243
7.5 Limited Types 244
7.6 Assignment and Finalization 246
7.6.1 Completion and Finalization 249
8 Visibility Rules 253

8.1 Declarative Region 253
8.2 Scope of Declarations 254
8.3 Visibility 255
8.3.1 Overriding Indicators 258
8.4 Use Clauses 259
8.5 Renaming Declarations 260
8.5.1 Object Renaming Declarations 260
8.5.2 Exception Renaming Declarations 262
8.5.3 Package Renaming Declarations 262
8.5.4 Subprogram Renaming Declarations 263
8.5.5 Generic Renaming Declarations 265
8.6 The Context of Overload Resolution 265
9 Tasks and Synchronization 269
9.1 Task Units and Task Objects 269
9.2 Task Execution - Task Activation 272
9.3 Task Dependence - Termination of Tasks 273
9.4 Protected Units and Protected Objects 275
9.5 Intertask Communication 278
9.5.1 Protected Subprograms and Protected Actions 283
9.5.2 Entries and Accept Statements 284
9.5.3 Entry Calls 287
9.5.4 Requeue Statements 290
9.6 Delay Statements, Duration, and Time 292
9.6.1 Formatting, Time Zones, and other operations for Time 295
9.7 Select Statements 301
9.7.1 Selective Accept 302
9.7.2 Timed Entry Calls 304
9.7.3 Conditional Entry Calls 305
9.7.4 Asynchronous Transfer of Control 305
9.8 Abort of a Task - Abort of a Sequence of Statements 307
9.9 Task and Entry Attributes 308
9.10 Shared Variables 309
9.10.1 Conflict Check Policies 310
9.11 Example of Tasking and Synchronization 312
10 Program Structure and Compilation Issues 315
10.1 Separate Compilation 315
10.1.1 Compilation Units - Library Units 315
10.1.2 Context Clauses - With Clauses 318
10.1.3 Subunits of Compilation Units 320
10.1.4 The Compilation Process 322
10.1.5 Pragmas and Program Units 323
10.1.6 Environment-Level Visibility Rules 324
10.2 Program Execution 324
10.2.1 Elaboration Control 327
11 Exceptions 333
11.1 Exception Declarations 333
11.2 Exception Handlers 334
11.3 Raise Statements and Raise Expressions 335
11.4 Exception Handling 336
11.4.1 The Package Exceptions 336
11.4.2 Pragmas Assert and Assertion_Policy 339

11.4.3 Example of Exception Handling 341
11.5 Suppressing Checks 343
11.6 Exceptions and Optimization 346
12 Generic Units 349
12.1 Generic Declarations 349
12.2 Generic Bodies 351
12.3 Generic Instantiation 352
12.4 Formal Objects 354
12.5 Formal Types 356
12.5.1 Formal Private and Derived Types 357
12.5.2 Formal Scalar Types 360
12.5.3 Formal Array Types 360
12.5.4 Formal Access Types 361
12.5.5 Formal Interface Types 362
12.6 Formal Subprograms 362
12.7 Formal Packages 365
12.8 Example of a Generic Package 367
13 Representation Issues 369
13.1 Operational and Representation Aspects 369
13.1.1 Aspect Specifications 373
13.2 Packed Types 377
13.3 Operational and Representation Attributes 377
13.4 Enumeration Representation Clauses 385
13.5 Record Layout 387
13.5.1 Record Representation Clauses 387
13.5.2 Storage Place Attributes 389
13.5.3 Bit Ordering 390
13.6 Change of Representation 391
13.7 The Package System 391
13.7.1 The Package System.Storage_Elements 393
13.7.2 The Package System.Address_To_Access_Conversions 394
13.8 Machine Code Insertions 395
13.9 Unchecked Type Conversions 396
13.9.1 Data Validity 397
13.9.2 The Valid Attribute 398
13.10 Unchecked Access Value Creation 399
13.11 Storage Management 399
13.11.1 Storage Allocation Attributes 403
13.11.2 Unchecked Storage Deallocation 404
13.11.3 Default Storage Pools 405
13.11.4 Storage Subpools 407
13.11.5 Subpool Reclamation 409
13.11.6 Storage Subpool Example 410
13.12 Pragma Restrictions and Pragma Profile 412
13.12.1 Language-Defined Restrictions and Profiles 414
13.13 Streams 416
13.13.1 The Streams Subsystem 416
13.13.2 Stream-Oriented Attributes 419
13.14 Freezing Rules 424
The Standard Libraries 427
Annex A (normative) Predefined Language Environment 429

A.1 The Package Standard 432
A.2 The Package Ada 436
A.3 Character Handling 436
A.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters 437
A.3.2 The Package Characters.Handling 437
A.3.3 The Package Characters.Latin_1 440
A.3.4 The Package Characters.Conversions 445
A.3.5 The Package Wide_Characters.Handling 447
A.3.6 The Package Wide_Wide_Characters.Handling 450
A.4 String Handling 450
A.4.1 The Package Strings 450
A.4.2 The Package Strings.Maps 450
A.4.3 Fixed-Length String Handling 454
A.4.4 Bounded-Length String Handling 462
A.4.5 Unbounded-Length String Handling 469
A.4.6 String-Handling Sets and Mappings 475
A.4.7 Wide_String Handling 476
A.4.8 Wide_Wide_String Handling 478
A.4.9 String Hashing 481
A.4.10 String Comparison 482
A.4.11 String Encoding 484
A.4.12 Universal Text Buffers 490
A.5 The Numerics Packages 492
A.5.1 Elementary Functions 493
A.5.2 Random Number Generation 496
A.5.3 Attributes of Floating Point Types 501
A.5.4 Attributes of Fixed Point Types 506
A.5.5 Big Numbers 507
A.5.6 Big Integers 507
A.5.7 Big Reals 509
A.6 Input-Output 511
A.7 External Files and File Objects 511
A.8 Sequential and Direct Files 513
A.8.1 The Generic Package Sequential_IO 513
A.8.2 File Management 515
A.8.3 Sequential Input-Output Operations 517
A.8.4 The Generic Package Direct_IO 518
A.8.5 Direct Input-Output Operations 519
A.9 The Generic Package Storage_IO 520
A.10 Text Input-Output 521
A.10.1 The Package Text_IO 522
A.10.2 Text File Management 529
A.10.3 Default Input, Output, and Error Files 530
A.10.4 Specification of Line and Page Lengths 531
A.10.5 Operations on Columns, Lines, and Pages 532
A.10.6 Get and Put Procedures 535
A.10.7 Input-Output of Characters and Strings 536
A.10.8 Input-Output for Integer Types 539
A.10.9 Input-Output for Real Types 540
A.10.10 Input-Output for Enumeration Types 543
A.10.11 Input-Output for Bounded Strings 544
A.10.12 Input-Output for Unbounded Strings 546
A.11 Wide Text Input-Output and Wide Wide Text Input-Output 547

A.12 Stream Input-Output **548**
A.12.1 The Package Streams.Stream_IO **548**
A.12.2 The Package Text_IO.Text_Streams **551**
A.12.3 The Package Wide_Text_IO.Text_Streams **551**
A.12.4 The Package Wide_Wide_Text_IO.Text_Streams **552**
A.13 Exceptions in Input-Output **552**
A.14 File Sharing **553**
A.15 The Package Command_Line **554**
A.15.1 The Packages Wide_Command_Line and Wide_Wide_Command_Line **555**
A.16 The Package Directories **555**
A.16.1 The Package Directories.Hierarchical_File_Names **563**
A.16.2 The Packages Wide_Directories and Wide_Wide_Directories **565**
A.17 The Package Environment_Variables **565**
A.17.1 The Packages Wide_Environment_Variables and Wide_Wide_Environment_Variables **567**
A.18 Containers **567**
A.18.1 The Package Containers **569**
A.18.2 The Generic Package Containers.Vectors **569**
A.18.3 The Generic Package Containers.Doubly_Linked_Lists **605**
A.18.4 Maps **627**
A.18.5 The Generic Package Containers.Hashed_Maps **637**
A.18.6 The Generic Package Containers.Ordered_Maps **646**
A.18.7 Sets **657**
A.18.8 The Generic Package Containers.Hashed_Sets **668**
A.18.9 The Generic Package Containers.Ordered_Sets **679**
A.18.10 The Generic Package Containers.Multiway_Trees **691**
A.18.11 The Generic Package Containers.Indefinite_Vectors **729**
A.18.12 The Generic Package Containers.Indefinite_Doubly_Linked_Lists **729**
A.18.13 The Generic Package Containers.Indefinite_Hashed_Maps **730**
A.18.14 The Generic Package Containers.Indefinite_Ordered_Maps **731**
A.18.15 The Generic Package Containers.Indefinite_Hashed_Sets **731**
A.18.16 The Generic Package Containers.Indefinite_Ordered_Sets **732**
A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees **732**
A.18.18 The Generic Package Containers.Indefinite_Holders **733**
A.18.19 The Generic Package Containers.Bounded_Vectors **738**
A.18.20 The Generic Package Containers.Bounded_Doubly_Linked_Lists **739**
A.18.21 The Generic Package Containers.Bounded_Hashed_Maps **741**
A.18.22 The Generic Package Containers.Bounded_Ordered_Maps **743**
A.18.23 The Generic Package Containers.Bounded_Hashed_Sets **744**
A.18.24 The Generic Package Containers.Bounded_Ordered_Sets **746**
A.18.25 The Generic Package Containers.Bounded_Multiway_Trees **748**
A.18.26 Array Sorting **750**
A.18.27 The Generic Package Containers.Synchronized_Queue_Interfaces **752**
A.18.28 The Generic Package Containers.Unbounded_Synchronized_Queues **753**
A.18.29 The Generic Package Containers.Bounded_Synchronized_Queues **754**
A.18.30 The Generic Package Containers.Unbounded_Priority_Queues **755**
A.18.31 The Generic Package Containers.Bounded_Priority_Queues **756**
A.18.32 The Generic Package Containers.Bounded_Indefinite_Holders **757**
A.18.33 Example of Container Use **758**
A.19 The Package Locales **760**
Annex B (normative) **Interface to Other Languages** **763**
B.1 Interfacing Aspects **763**

B.2 The Package Interfaces 767
B.3 Interfacing with C and C++ 768
B.3.1 The Package Interfaces.C.Strings 775
B.3.2 The Generic Package Interfaces.C.Pointers 778
B.3.3 Unchecked Union Types 781
B.4 Interfacing with COBOL 783
B.5 Interfacing with Fortran 790
Annex C (normative) Systems Programming 793
C.1 Access to Machine Operations 793
C.2 Required Representation Support 794
C.3 Interrupt Support 794
C.3.1 Protected Procedure Handlers 796
C.3.2 The Package Interrupts 798
C.4 Preelaboration Requirements 800
C.5 Aspect Discard_Names 801
C.6 Shared Variable Control 802
C.6.1 The Package System.Atomic_Operations 806
C.6.2 The Package System.Atomic_Operations.Exchange 806
C.6.3 The Package System.Atomic_Operations.Test_and_Set 807
C.6.4 The Package System.Atomic_Operations.Integer_Arithmetic 808
C.6.5 The Package System.Atomic_Operations.Modular_Arithmetic 809
C.7 Task Information 811
C.7.1 The Package Task_Identification 811
C.7.2 The Package Task_Attributes 812
C.7.3 The Package Task_Termination 815
Annex D (normative) Real-Time Systems 817
D.1 Task Priorities 817
D.2 Priority Scheduling 819
D.2.1 The Task Dispatching Model 820
D.2.2 Task Dispatching Pragmas 822
D.2.3 Preemptive Dispatching 823
D.2.4 Non-Preemptive Dispatching 824
D.2.5 Round Robin Dispatching 825
D.2.6 Earliest Deadline First Dispatching 827
D.3 Priority Ceiling Locking 830
D.4 Entry Queuing Policies 832
D.4.1 Admission Policies 834
D.5 Dynamic Priorities 834
D.5.1 Dynamic Priorities for Tasks 835
D.5.2 Dynamic Priorities for Protected Objects 836
D.6 Preemptive Abort 837
D.7 Tasking Restrictions 838
D.8 Monotonic Time 841
D.9 Delay Accuracy 844
D.10 Synchronous Task Control 845
D.10.1 Synchronous Barriers 846
D.11 Asynchronous Task Control 847
D.12 Other Optimizations and Determinism Rules 848
D.13 The Ravenscar and Jorvik Profiles 849
D.14 Execution Time 851
D.14.1 Execution Time Timers 853
D.14.2 Group Execution Time Budgets 855

D.14.3 Execution Time of Interrupt Handlers....857
D.15 Timing Events....858
D.16 Multiprocessor Implementation....860
D.16.1 Multiprocessor Dispatching Domains....861
Annex E (normative) Distributed Systems....865
E.1 Partitions....865
E.2 Categorization of Library Units....866
E.2.1 Shared Passive Library Units....867
E.2.2 Remote Types Library Units....868
E.2.3 Remote Call Interface Library Units....869
E.3 Consistency of a Distributed System....871
E.4 Remote Subprogram Calls....871
E.4.1 Asynchronous Remote Calls....873
E.4.2 Example of Use of a Remote Access-to-Class-Wide Type....874
E.5 Partition Communication Subsystem....876
Annex F (normative) Information Systems....879
F.1 Machine_Radix Attribute Definition Clause....879
F.2 The Package Decimal....880
F.3 Edited Output for Decimal Types....881
F.3.1 Picture String Formation....882
F.3.2 Edited Output Generation....886
F.3.3 The Package Text_IO.Editing....890
F.3.4 The Package Wide_Text_IO.Editing....893
F.3.5 The Package Wide_Wide_Text_IO.Editing....893
Annex G (normative) Numerics....895
G.1 Complex Arithmetic....895
G.1.1 Complex Types....895
G.1.2 Complex Elementary Functions....900
G.1.3 Complex Input-Output....903
G.1.4 The Package Wide_Text_IO.Complex_IO....906
G.1.5 The Package Wide_Wide_Text_IO.Complex_IO....906
G.2 Numeric Performance Requirements....906
G.2.1 Model of Floating Point Arithmetic....907
G.2.2 Model-Oriented Attributes of Floating Point Types....908
G.2.3 Model of Fixed Point Arithmetic....909
G.2.4 Accuracy Requirements for the Elementary Functions....911
G.2.5 Performance Requirements for Random Number Generation....913
G.2.6 Accuracy Requirements for Complex Arithmetic....913
G.3 Vector and Matrix Manipulation....915
G.3.1 Real Vectors and Matrices....915
G.3.2 Complex Vectors and Matrices....920
Annex H (normative) High Integrity Systems....933
H.1 Pragma Normalize_Scalars....933
H.2 Documentation of Implementation Decisions....934
H.3 Reviewable Object Code....934
H.3.1 Pragma Reviewable....934
H.3.2 Pragma Inspection_Point....935
H.4 High Integrity Restrictions....936
H.4.1 Aspect No_Controlled_Parts....939
H.5 Pragma Detect_Blocking....939

H.6 Pragma Partition_Elaboration_Policy 940
H.7 Extensions to Global and Global'Class Aspects 941
H.7.1 The Use_Formal and Dispatching Aspects 942
Annex J (normative) Obsolescent Features 945
J.1 Renamings of Library Units 945
J.2 Allowed Replacements of Characters 945
J.3 Reduced Accuracy Subtypes 946
J.4 The Constrained Attribute 947
J.5 ASCII 947
J.6 Numeric_Error 948
J.7 At Clauses 948
J.7.1 Interrupt Entries 948
J.8 Mod Clauses 949
J.9 The Storage_Size Attribute 950
J.10 Specific Suppression of Checks 950
J.11 The Class Attribute of Untagged Incomplete Types 951
J.12 Pragma Interface 951
J.13 Dependence Restriction Identifiers 951
J.14 Character and Wide_Character Conversion Functions 951
J.15 Aspect-related Pragmas 952
J.15.1 Pragma Inline 953
J.15.2 Pragma No_Return 953
J.15.3 Pragma Pack 953
J.15.4 Pragma Storage_Size 954
J.15.5 Interfacing Pragmas 954
J.15.6 Pragma Unchecked_Union 955
J.15.7 Pragmas Interrupt_Handler and Attach_Handler 955
J.15.8 Shared Variable Pragmas 956
J.15.9 Pragma CPU 957
J.15.10 Pragma Dispatching_Domain 957
J.15.11 Pragmas Priority and Interrupt_Priority 958
J.15.12 Pragma Relative_Deadline 958
J.15.13 Pragma Asynchronous 959
J.15.14 Elaboration Control Pragmas 959
J.15.15 Distribution Pragmas 960
Annex K (informative) Language-Defined Aspects and Attributes 961
K.1 Language-Defined Aspects 961
K.2 Language-Defined Attributes 965
Annex L (informative) Language-Defined Pragmas 985
Annex M (informative) Summary of Documentation Requirements 989
M.1 Specific Documentation Requirements 989
M.2 Implementation-Defined Characteristics 991
M.3 Implementation Advice 998
Annex N (informative) Glossary 1009
Annex P (informative) Syntax Summary 1011
P.1 Syntax Rules 1011
P.2 Syntax Cross Reference 1036
Annex Q (informative) Language-Defined Entities 1051

Q.1 Language-Defined Packages 1051
Q.2 Language-Defined Types and Subtypes 1054
Q.3 Language-Defined Subprograms 1059
Q.4 Language-Defined Exceptions 1073
Q.5 Language-Defined Objects 1074
Index 1079

Foreword

This document is the Ada Reference Manual. The International Standard for the programming language Ada is ISO/IEC 8652:2023(E). The International Standard is derived from the Ada Reference Manual, with various non-normative changes. In particular, the International Standard numbers clauses differently, omits paragraph numbers, eliminates the Acknowledgments, and modifies various front matter such as the Title page and the Foreword. 0.1/5

The Ada Working Group ISO/IEC JTC 1/SC 22/WG 9 is tasked by ISO with the work item to interpret and maintain the International Standard and to produce Technical Corrigenda, as appropriate. The technical work on the International Standard is performed by the Ada Rapporteur Group (ARG) of WG 9. 0.2/4

AXE Consultants produces the Ada Reference Manual in consultation with the ARG, along with drafts of other documents as needed. ISO/IEC documents often list the individual changes that need to be made to the text of a Standard, rather than simply updating the document. As such, an International Standard is often found in several parts, while the Ada Reference Manual is always a single document. 0.3/2

In June 2016, WG 9 approved a tentative schedule for the preparation of an Amendment or Revision to the International Standard, with a delivery no earlier than 2018. In July 2019, WG 9 approved an additional review and prototyping period for this revision, extending the delivery to no earlier than late 2020. The draft standard was delivered to WG 9 in July 2021, and the Standard was published as ISO/IEC 8652:2023 in May 2023. This edition is informally known as Ada 2022. 0.5/6

Since the completion of Ada 2022, work has continued on Ada, both to correct errors and to add enhancements. A Corrigendum to correct errors and add missing rules was submitted to standardization in June 2025; it was rejected for process reasons and replaced by an Amendment with similar contents as well as several minor enhancements in December 2025. As of this writing it is unknown whether standardization will succeed. However, it is valuable to have all of the changes available in one place. Thus, we've created this consolidated edition. 0.6/6

This version of the Ada Reference Manual shows the current state of Ada 2022 including the changes made by Amendment 1. 1/6

Significant changes in this consolidated edition are as follows: 2/6

- Automatic construction of functions as provided for null extensions is now optionally available for most type extensions. 3/6
- Rounding attributes similar to the ones long available for floating point types are added for fixed point types. 4/6
- Clarified that both user-defined literals and container aggregates can be of class-wide types, and described what happens when they do have such types. 5/6
- Additional capabilities are added to many language-defined packages, adding important missing functionality. 6/6
- Minor extensions are made to make the language more regular and improve usability. 7/6
- Many corrections and clarifications have been made to the original edition of Ada 2022. 8/6

Introduction

1 This is the Ada Reference Manual.

2 Other available Ada documents include:

3/5 • Ada 2022 Overview. This gives an introduction to the changes and new features in Ada 2022. Programmers should read this overview before reading Reference Manual in depth.

4/1 • *This paragraph was deleted.*

5/3 • The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the consolidated Ada Reference Manual, plus various annotations. It is intended primarily for compiler writers, validation test writers, and others who wish to study the fine details. The annotations include detailed rationale for individual rules and explanations of some of the more arcane interactions among the rules.

Design Goals

6/6 Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. The 1995 revision to the language was designed to provide greater flexibility and extensibility, additional control over storage management and synchronization, and standardized packages oriented toward supporting important application areas, while at the same time retaining the original emphasis on reliability, maintainability, and efficiency. Subsequent editions, including this fourth edition, have provided further flexibility and added more standardized packages within the framework provided by the 1995 revision.

7 The need for languages that promote reliability and simplify maintenance is well established. Hence emphasis was placed on program readability over ease of writing. For example, the rules of the language require that program variables be explicitly declared and that their type be specified. Since the type of a variable is invariant, compilers can ensure that operations on variables are compatible with the properties intended for objects of the type. Furthermore, error-prone notations have been avoided, and the syntax of the language avoids the use of encoded forms in favor of more English-like constructs. Finally, the language offers support for separate compilation of program units in a way that facilitates program development and maintenance, and which provides the same degree of checking between units as within a unit.

8 Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep to a relatively small number of underlying concepts integrated in a consistent and systematic way while continuing to avoid the pitfalls of excessive involution. The design especially aims to provide language constructs that correspond intuitively to the normal expectations of users.

9 Like many other human activities, the development of programs is becoming ever more decentralized and distributed. Consequently, the ability to assemble a program from independently produced software components continues to be a central idea in the design. The concepts of packages, of private types, and of generic units are directly related to this idea, which has ramifications in many other aspects of the language. An allied concern is the maintenance of programs to match changing requirements; type extension and the hierarchical library enable a program to be modified while minimizing disturbance to existing tested and trusted components.

10/5 No language can avoid the problem of efficiency. Languages that require over-elaborate compilers, or that lead to the inefficient use of storage or execution time, force these inefficiencies on all machines and on all programs. Every construct of the language was examined in the light of present implementation

techniques. Any proposed construct whose implementation was unclear or that required excessive machine resources was rejected. Parallel constructs were introduced to simplify making safe and efficient use of modern multicore architectures.

Language Summary

An Ada program is composed of one or more program units. Program units can be subprograms (which define executable algorithms), packages (which define collections of entities), task units (which define concurrent computations), protected units (which define operations for the coordinated sharing of data between tasks), or generic units (which define parameterized forms of packages and subprograms). Each program unit normally consists of two parts: a specification, containing the information that is visible to other units, and a body, containing the implementation details, which are not visible to other units. Most program units can be compiled separately. 11/5

This distinction of the specification and body, and the ability to compile units separately, allows a program to be designed, written, and tested as a set of largely independent software components. 12

An Ada program will normally make use of a library of program units of general utility. The language provides means whereby individual organizations can construct their own libraries. All libraries are structured in a hierarchical manner; this enables the logical decomposition of a subsystem into individual components. The text of a separately compiled program unit names the library units it requires. 13/5

Program Units 14

A subprogram is the basic unit for expressing an algorithm. There are two kinds of subprograms: procedures and functions. A procedure is the means of invoking a series of actions. For example, it can read data, update variables, or produce some output. It can have parameters, to provide a controlled means of passing information between the procedure and the point of call. A function is the means of invoking the computation of a value. It is similar to a procedure, but in addition will return a result. 15/5

A package is the basic unit for defining a collection of logically related entities. For example, a package can be used to define a set of type declarations and associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package specification. 16

Subprogram and package units can be compiled separately and arranged in hierarchies of parent and child units giving fine control over visibility of the logical properties and their detailed implementation. 17/5

A task unit is the basic unit for defining a task whose sequence of actions can be executed concurrently with those of other tasks. Such tasks can be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor. A task unit can define either a single executing task or a task type permitting the creation of any number of similar tasks. 18/5

A protected unit is the basic unit for defining protected operations for the coordinated use of data shared between tasks. Simple mutual exclusion is provided automatically, and more elaborate sharing protocols can be defined. A protected operation can either be a subprogram or an entry. A protected entry specifies a Boolean expression (an entry barrier) that blocks the execution of the body until it evaluates to True. A protected unit can define a single protected object or a protected type permitting the creation of several similar objects. 19/5

20 *Declarations and Statements*

21 The body of a program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a sequence of statements, which defines the execution of the program unit.

22/5 The declarative part associates names with declared entities. For example, a name can denote a type, a constant, a variable, or an exception. A declarative part also introduces the names and parameters of other nested subprograms, packages, task units, protected units, and generic units to be used in the program unit.

23/5 The sequence of statements describes a sequence of actions to be performed. The statements are executed in succession (unless a transfer of control causes execution to continue from another place).

24 An assignment statement changes the value of a variable. A procedure call invokes execution of a procedure after associating any actual parameters provided at the call with the corresponding formal parameters.

25 Case statements and if statements allow the selection of an enclosed sequence of statements based on the value of an expression or on the value of a condition.

26/5 The loop statement provides the basic iterative mechanism in the language. A loop statement specifies a sequence of statements that are executed repeatedly as directed by an iteration scheme, or until an exit statement is encountered.

27 A block statement comprises a sequence of statements preceded by the declaration of local entities used by the statements.

28/5 Certain statements are associated with concurrent execution. A delay statement delays the execution of a task for a specified duration or until a specified time. An entry call statement is written as a procedure call statement; it requests an operation on a task or on a protected object, blocking the caller until the operation can be performed. A called task can accept an entry call by executing a corresponding accept statement, which specifies the actions then to be performed as part of the rendezvous with the calling task. An entry call on a protected object is processed when the corresponding entry barrier evaluates to true, whereupon the body of the entry is executed. The requeue statement permits the provision of a service as a number of related activities with preference control. One form of the select statement allows a selective wait for one of several alternative rendezvous. Other forms of the select statement allow conditional or timed entry calls and the asynchronous transfer of control in response to some triggering event. Various parallel constructs, including parallel loops and parallel blocks, support the initiation of multiple logical threads of control designed to execute in parallel when multiple processors are available.

29/5 Execution of a program unit can encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation can exceed the maximum allowed value of a number, or an attempt can be made to access an array component by using an incorrect index value. To deal with such error situations, the statements of a program unit can be textually followed by exception handlers that specify the actions to be taken when the error situation arises. Exceptions can be raised explicitly by a raise statement.

30 *Data Types*

31/5 Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main categories of types are elementary types (comprising enumeration, numeric, and access types) and composite types (including array and record types).

An enumeration type defines an ordered set of distinct enumeration literals, for example a list of states or an alphabet of characters. The enumeration types Boolean, Character, Wide_Character, and Wide_Wide_Character are predefined. 32/2

Numeric types provide a means of performing exact or approximate numerical computations. Exact computations use integer types, which denote sets of consecutive integers. Approximate computations use either fixed point types, with absolute bounds on the error, or floating point types, with relative bounds on the error. The numeric types Integer, Float, and Duration are predefined. 33

Composite types allow definitions of structured objects with related components. The composite types in the language include arrays and records. An array is an object with indexed components of the same type. A record is an object with named components of possibly different types. Task and protected types are also forms of composite types. The array types String, Wide_String, and Wide_Wide_String are predefined. 34/2

Record, task, and protected types can have special components called discriminants which parameterize the type. Variant record structures that depend on the values of discriminants can be defined within a record type. 35/5

Access types allow the construction of linked data structures. A value of an access type represents a reference to an object declared as aliased or to an object created by the evaluation of an allocator. Several variables of an access type can designate the same object, and components of one object can designate the same or other objects. Both the elements in such linked data structures and their relation to other elements can be altered during program execution. Access types also permit references to subprograms to be stored, passed as parameters, and ultimately dereferenced as part of an indirect call. 36/5

Private types permit restricted views of a type. A private type can be defined in a package so that only the logically necessary properties are made visible to the users of the type. The full structural details that are externally irrelevant are then only available within the package and any child units. 37

From any type a new type can be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations can be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives can be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension are tagged, so that the specific type of an operand within a derivation class can be identified at run time. When an operation of a tagged type is applied to an operand whose specific type is not known until run time, implicit dispatching is performed based on the tag of the operand. 38/5

Interface types provide abstract models from which other interfaces and types can be composed and derived. This provides a reliable form of multiple inheritance. Interface types can also be implemented by task types and protected types thereby enabling concurrent programming and inheritance to be merged. 38.1/5

The concept of a type is further refined by the concept of a subtype, whereby a user can constrain the set of allowed values of a type. Subtypes can be used to define subranges of scalar types, arrays with a limited set of index values, and records and private types with particular discriminant values. 39

Other Facilities 40

Aspect clauses can be used to specify the mapping between types and features of an underlying machine. For example, the user can specify that objects of a given type are to be represented with a given number of bits, or that the components of a record are to be represented using a given storage layout. Other features allow the controlled use of low level, nonportable, or implementation-dependent aspects, including the direct insertion of machine code. 41/5

41.1/5 Aspect clauses can also be used to specify more abstract properties of program entities, such as the pre- and postconditions of a subprogram, or the invariant for a private type. Additional aspects are specifiable to allow user-defined types to use constructs of the language, such as literals, aggregates, or indexing, normally reserved for particular language-defined categories of types, such as numeric types, record types, or array types.

42/2 The predefined environment of the language provides for input-output and other capabilities by means of standard library packages. Input-output is supported for values of user-defined as well as of predefined types. Standard means of representing values in display form are also provided.

42.1/2 The predefined standard library packages provide facilities such as string manipulation, containers of various kinds (vectors, lists, maps, etc.), mathematical functions, random number generation, and access to the execution environment.

42.2/2 The specialized annexes define further predefined library packages and facilities with emphasis on areas such as real-time scheduling, interrupt handling, distributed systems, numerical computation, and high-integrity systems.

43 Finally, the language provides a powerful means of parameterization of program units, called generic program units. The generic parameters can be types and subprograms (as well as objects and packages) and so allow general algorithms and data structures to be defined that are applicable to all types of a given class.

Paragraphs 44 through 57 have been replaced and moved to the Foreword.

Instructions for Comment Submission

57.22/5 Informal comments on this Reference Manual can be submitted in three ways:

57.23/5 - via e-mail to:

57.24/5 **ada-comment@ada-auth.org**

57.25/5 - by filling out a web form at:

57.26/5 **https://arg.adaic.org/community-input**

57.27/5 - by creating a new “issue” on the ARG GitHub site:

57.28/5 **https://github.com/Ada-Rapporteur-Group/User-Community-Input**

58/5 If appropriate, the Project Editor will initiate the defect correction procedure.

59 Comments should use the following format:

60/5
!topic *Title summarizing comment*
!reference Ada 2022 RM*ss.ss(pp)*
!from *Author Name yy-mm-dd*
!keywords *keywords related to topic*
!discussion

text of discussion

61/3 where *ss.ss* is the clause or subclause number, *pp* is the paragraph number where applicable, and *yy-mm-dd* is the date the comment was sent. The date is optional, as is the **!keywords** line.

Please use a descriptive “Subject” when sending an e-mail message, or title when creating a new issue, and limit each message to a single comment. 62/5

When correcting typographical errors or making minor wording suggestions, please put the correction directly as the topic of the comment; use square brackets [ ] to indicate text to be omitted and curly braces { } to indicate text to be added, and provide enough context to make the nature of the suggestion self-evident or put additional information in the body of the comment, for example: 63

!topic [c]{C}haracter 64
!topic it[']s meaning is not defined

This paragraph was deleted. 65/5

Acknowledgments for the Ada 83 edition

Ada is the result of a collective effort to design a common language for programming large scale and real-time systems. 65.1/3

The common high order language program began in 1974. The requirements of the United States Department of Defense were formalized in a series of documents which were extensively reviewed by the Services, industrial organizations, universities, and foreign military departments. The Ada language was designed in accordance with the final (1978) form of these requirements, embodied in the Steelman specification. 65.2/3

The Ada design team was led by Jean D. Ichbiah and has included Bernd Krieg-Brueckner, Brian A. Wichmann, Henry F. Ledgard, Jean-Claude Heliard, Jean-Loup Gailly, Jean-Raymond Abrial, John G.P. Barnes, Mike Woodger, Olivier Roubine, Paul N. Hilfinger, and Robert Firth. 65.3/3

At various stages of the project, several people closely associated with the design team made major contributions. They include J.B. Goodenough, R.F. Brender, M.W. Davis, G. Ferran, K. Lester, L. MacLaren, E. Morel, I.R. Nassi, I.C. Pyle, S.A. Schuman, and S.C. Vestal. 65.4/3

Two parallel efforts that were started in the second phase of this design had a deep influence on the language. One was the development of a formal definition using denotational semantics, with the participation of V. Donzeau-Gouge, G. Kahn, and B. Lang. The other was the design of a test translator with the participation of K. Ripken, P. Boullier, P. Cadiou, J. Holden, J.F. Hueras, R.G. Lange, and D.T. Cornhill. The entire effort benefitted from the dedicated assistance of Lyn Churchill and Marion Myers, and the effective technical support of B. Gravem, W.L. Heimerdinger, and P. Cleve. H.G. Schmitz served as program manager. 65.5/3

Over the five years spent on this project, several intense week-long design reviews were conducted, with the participation of P. Belmont, B. Brosgol, P. Cohen, R. Dewar, A. Evans, G. Fisher, H. Harte, A.L. Hisgen, P. Knueven, M. Kronental, N. Lomuto, E. Ploedereder, G. Seegmueller, V. Stenning, D. Taffs, and also F. Belz, R. Converse, K. Correll, A.N. Habermann, J. Sammet, S. Squires, J. Teller, P. Wegner, and P.R. Wetherall. 65.6/3

Several persons had a constructive influence with their comments, criticisms and suggestions. They include P. Brinch Hansen, G. Goos, C.A.R. Hoare, Mark Rain, W.A. Wulf, and also E. Boebert, P. Bonnard, H. Clausen, M. Cox, G. Dismukes, R. Eachus, T. Froggatt, H. Ganzinger, C. Hewitt, S. Kamin, R. Kotler, O. Lecarme, J.A.N. Lee, J.L. Mansion, F. Minel, T. Phinney, J. Roehrich, V. Schneider, A. Singer, D. Slosberg, I.C. Wand, the reviewers of Ada-Europe, AdaTech, Afcet, those of the LMSC review team, and those of the Ada Tokyo Study Group. 65.7/3

65.8/3 These reviews and comments, the numerous evaluation reports received at the end of the first and second phase, the nine hundred language issue reports and test and evaluation reports received from fifteen different countries during the third phase of the project, the thousands of comments received during the ANSI Canvass, and the on-going work of the IFIP Working Group 2.4 on system implementation languages and that of the Purdue Europe LTPL-E committee, all had a substantial influence on the final definition of Ada.

65.9/3 The Military Departments and Agencies have provided a broad base of support including funding, extensive reviews, and countless individual contributions by the members of the High Order Language Working Group and other interested personnel. In particular, William A. Whitaker provided leadership for the program during the formative stages. David A. Fisher was responsible for the successful development and refinement of the language requirement documents that led to the Steelman specification.

65.10/3 The Ada 83 language definition was developed by Cii Honeywell Bull and later Alsys, and by Honeywell Systems and Research Center, under contract to the United States Department of Defense. William E. Carlson and later Larry E. Druffel served as the technical representatives of the United States Government and effectively coordinated the efforts of all participants in the Ada program.

Acknowledgments for the Ada 95 edition

66 This Reference Manual was prepared by the Ada 9X Mapping/Revision Team based at Intermetrics, Inc., which has included: W. Carlson, Program Manager; T. Taft, Technical Director; J. Barnes (consultant); B. Brosgol (consultant); R. Duff (Oak Tree Software); M. Edwards; C. Garrity; R. Hilliard; O. Pazy (consultant); D. Rosenfeld; L. Shafer; W. White; M. Woodger.

67 The following consultants to the Ada 9X Project contributed to the Specialized Needs Annexes: T. Baker (Real-Time/Systems Programming — SEI, FSU); K. Dritz (Numerics — Argonne National Laboratory); A. Gargaro (Distributed Systems — Computer Sciences); J. Goodenough (Real-Time/Systems Programming — SEI); J. McHugh (Secure Systems — consultant); B. Wichmann (Safety-Critical Systems — NPL: UK).

68 This work was regularly reviewed by the Ada 9X Distinguished Reviewers and the members of the Ada 9X Rapporteur Group (XRG): E. Ploedereder, Chairman of DRs and XRG (University of Stuttgart: Germany); B. Bardin (Hughes); J. Barnes (consultant: UK); B. Brett (DEC); B. Brosgol (consultant); R. Brukardt (RR Software); N. Cohen (IBM); R. Dewar (NYU); G. Dismukes (TeleSoft); A. Evans (consultant); A. Gargaro (Computer Sciences); M. Gerhardt (ESL); J. Goodenough (SEI); S. Heilbrunner (University of Salzburg: Austria); P. Hilfinger (UC/Berkeley); B. Källberg (CelsiusTech: Sweden); M. Kamrad II (Unisys); J. van Katwijk (Delft University of Technology: The Netherlands); V. Kaufman (Russia); P. Kruchten (Rational); R. Landwehr (CCI: Germany); C. Lester (Portsmouth Polytechnic: UK); L. Månsson (TELIA Research: Sweden); S. Michell (Multiprocessor Toolsmiths: Canada); M. Mills (US Air Force); D. Pogge (US Navy); K. Power (Boeing); O. Roubine (Verdix: France); A. Strohmeier (Swiss Fed Inst of Technology: Switzerland); W. Taylor (consultant: UK); J. Tokar (Tartan); E. Vasilescu (Grumman); J. Vladik (Prospeks s.r.o.: Czech Republic); S. Van Vlierberghe (OFFIS: Belgium).

69 Other valuable feedback influencing the revision process was provided by the Ada 9X Language Precision Team (Odyssey Research Associates), the Ada 9X User/Implementer Teams (AETECH, Tartan, TeleSoft), the Ada 9X Implementation Analysis Team (New York University) and the Ada community-at-large.

70 Special thanks go to R. Mathis, Convenor of ISO/IEC JTC 1/SC 22 Working Group 9.

71 The Ada 9X Project was sponsored by the Ada Joint Program Office. Christine M. Anderson at the Air Force Phillips Laboratory (Kirtland AFB, NM) was the project manager.

Acknowledgments for the Corrigendum version

The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this update possible. 71.1/3

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, E. Ploedereder (Germany), who kept the process moving; J. Barnes (UK) and K. Ishihata (Japan), whose extremely detailed reviews kept the editor on his toes; G. Dismukes (USA), M. Kamrad (USA), P. Leroy (France), S. Michell (Canada), T. Taft (USA), J. Tokar (USA), and other members too numerous to mention. 71.2/1

Special thanks go to R. Duff (USA) for his explanations of the previous system of formatting of these documents during the tedious conversion to more modern formats. Special thanks also go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), without whose help and support the Corrigendum and this consolidated reference manual would not have been possible. 71.3/1

Acknowledgments for the Amendment 1 version

The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this update possible. 71.4/3

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording corrections was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, P. Leroy (France), who kept the process on schedule; J. Barnes (UK) whose careful reviews found many typographical errors; T. Taft (USA), who always seemed to have a suggestion when we were stuck, and who also was usually able to provide the valuable service of explaining why things were as they are; S. Baird (USA), who found many obscure problems with the proposals; and A. Burns (UK), who pushed many of the real-time proposals to completion. Other ARG members who contributed were: R. Dewar (USA), G. Dismukes (USA), R. Duff (USA), K. Ishihata (Japan), S. Michell (Canada), E. Ploedereder (Germany), J.P. Rosen (France), E. Schonberg (USA), J. Tokar (USA), and T. Vardanega (Italy). 71.5/2

Special thanks go to Ada-Europe and the Ada Resource Association, without whose help and support the Amendment and this consolidated reference manual would not have been possible. M. Heaney (USA) requires special thanks for his tireless work on the containers packages. Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Moore (USA), who guided the document through the standardization process. 71.6/2

Acknowledgments for the Ada 2012 edition

The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this revision possible. 71.7/3

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording changes was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, E. Schonberg (USA), who guided the work; T. Taft (USA), whose insights broke many logjams, both in design and wording; J. Barnes (UK) whose careful reviews uncovered many editorial errors; S. Baird (USA), who repeatedly found obscure interactions with the proposals that the rest of us missed. Other ARG members who substantially contributed were: A. Burns (UK), J. Cousins (UK), R. Dewar (USA), G. Dismukes (USA), R. Duff (USA), P. Leroy (France), 71.8/3

B. Moore (Canada), E. Ploedereder (Germany), J.P. Rosen (France), B. Thomas (USA), and T. Vardanega (Italy).

71.9/3 Special thanks go to Ada-Europe and the Ada Resource Association, without whose help and support this third edition of the Ada Standard would not have been possible. A special mention has to go to A. Beneschan (USA) for his efforts in eliminating sloppiness in our wording. M. Heaney (USA) also deserves a mention for his efforts to improve the containers packages. Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Tokar (USA), who guided the document through the standardization process.

Acknowledgments for the Ada 2012 Corrigendum 1 version

71.10/4 The editor [R. Brukardt (USA)] would like to thank the many people whose hard work and assistance has made this update possible.

71.11/4 Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work on creating and editing the wording changes was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, J. Cousins (UK), who guided the work; T. Taft (USA), who seems to have the ability to cut any Gordian knot we encounter in wording; J. Barnes (UK) who continues to be able to find editorial errors invisible to most; S. Baird (USA), who so frequently finds obscure interactions that we now have named such things for him. Other ARG members who substantially contributed were: A. Burns (UK), R. Dewar (USA), G. Dismukes (USA), R. Duff (USA), B. Moore (Canada), E. Ploedereder (Germany), J.P. Rosen (France), E. Schonberg (USA), and T. Vardanega (Italy).

71.12/4 Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, J. Tokar (USA), who guided the document through the standardization process.

Acknowledgments for the Ada 2022 version

71.13/5 The editor [R. Brukardt] would like to thank the many people whose hard work and assistance has made this revision possible.

71.14/5 Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work in all steps of the process, from determining problems to address, reviewing feature designs, and creating and editing wording changes, was critical to the entire process. Especially valuable contributions came from the chairman of the ARG through June 2018, J. Cousins, who guided the work and ensured we followed defined procedures; his replacement as chairman, S. Baird, who ably powered through obstacles to complete the work while continuing to find obscure interactions; T. Taft, who often solved difficult problems that had stumped others; B. Moore, whose frequent suggestions for parallel constructs greatly improved the result. Other ARG members who substantially contributed were: R. Amiard, J. Barnes, A. Burns, A. Charlet, G. Dismukes, C. Dross, R. Duff, E. Fish, E. Ploedereder, J.P. Rosen, F. Schanda, E. Schonberg, J. Squirek, T. Vardanega, and R. Wai.

71.15/5 Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, P. Rogers, who guided the document through the standardization process.

Acknowledgments for the Ada 2022 Amendment version

71.16/6 The editor [R. Brukardt] would like to thank the many people whose hard work and assistance has made this update possible.

Thanks go out to all of the members of the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group, whose work in all steps of the process, from determining problems to address, reviewing feature designs, and creating and editing wording changes, was critical to the entire process. Especially valuable contributions came from the chairman of the ARG, S. Baird, who guided the work while continuing to find obscure interactions; and from T. Taft, who created many of the solutions in this Amendment. Other ARG members who substantially contributed were: J. Barnes, A. Charlet, J. Cousins, G. Dismukes, C. Dross, R. Duff, E. Fish, C. Grein, N. Holsti, B. Moore, J.P. Rosen, E. Schonberg, J. Squirek, T. Vardanega, and R. Wai. 71.17/6

Finally, special thanks go to the convenor of ISO/IEC JTC 1/SC 22/WG 9, P. Rogers, who guided the document through the standardization process. 71.18/6

Using this version of the Ada Reference Manual

This document has been revised with the changes specified in Amendment 1 for Ada 2022. In addition, a variety of editorial errors have been corrected. 72/6

Changes to the original 1995 version of the Ada Reference Manual can be identified by the version number following the paragraph number. Paragraphs with a version number of /1 were changed by Technical Corrigendum 1 for Ada 95 or were editorial corrections at that time, while paragraphs with a version number of /2 were changed by Amendment 1 or were more recent editorial corrections, and paragraphs with a version number of /3 were changed by the 2012 edition of the Reference Manual (including additional editorial corrections). Paragraphs with a version number of /4 were changed by Technical Corrigendum 1 for Ada 2012 or were editorial corrections at that time. Paragraphs with a version number of /5 were changed by the 2022 edition of the Reference Manual (including additional editorial corrections). Paragraphs with a version number of /6 were changed by Amendment 1 for Ada 2022 or were editorial corrections at that time. Paragraphs not so marked are unchanged since the original 1995 edition of the Ada Reference Manual, and have the same paragraph numbers as in that edition. In addition, some versions of this document include revision bars near the paragraph numbers. Where paragraphs are inserted, the paragraph numbers are of the form pp.nn, where pp is the number of the preceding paragraph, and nn is an insertion number. For instance, the first paragraph inserted after paragraph 8 is numbered 8.1, the second paragraph inserted is numbered 8.2, and so on. Deleted paragraphs are indicated by the text *This paragraph was deleted.* Deleted paragraphs include empty paragraphs that were numbered in the 1995 edition of the Ada Reference Manual. 73/6

1 General

1.1 Scope

This Reference Manual specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of computing systems. 1/3

1.1.1 Extent

This Reference Manual specifies: 1

- The form of a program written in Ada; 2
- The effect of translating and executing such a program; 3
- The manner in which program units can be combined to form Ada programs; 4/5
- The language-defined library units that a conforming implementation is required to supply; 5
- The permissible variations in conformance to the rules of this document, and the manner in which they are to be documented; 6/5
- Those violations of the requirements of this document that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program containing such violations; 7/5
- Those violations of the requirements of this document that a conforming implementation is not required to detect. 8/5

This Reference Manual does not specify: 9

- The means whereby a program written in Ada is transformed into object code executable by a processor; 10
- The means whereby translation or execution of programs is invoked and the executing units are controlled; 11
- The size or speed of the object code, or the relative execution speed of different language constructs; 12
- The form or contents of any listings produced by implementations; in particular, the form or contents of error or warning messages; 13
- The effect of unspecified execution; 14/5
- The size of a program or program unit that will exceed the capacity of a particular conforming implementation. 15

1.1.2 Structure

This document contains thirteen clauses, fifteen annexes, and an index. 1/3

The *core* of the Ada language consists of: 2

- Clauses 1 through 13 3/3
- Annex A, “Predefined Language Environment” 4
- Annex B, “Interface to Other Languages” 5

- Annex J, “Obsolescent Features”

The following *Specialized Needs Annexes* define features that are needed by certain application areas:

- Annex C, “Systems Programming”
- Annex D, “Real-Time Systems”
- Annex E, “Distributed Systems”
- Annex F, “Information Systems”
- Annex G, “Numerics”
- Annex H, “High Integrity Systems”

The core language and the Specialized Needs Annexes are normative, except that the material in each of the items listed below is informative:

- Text under a NOTES or Examples heading.
- Each subclause whose title starts with the word “Example” or “Examples”.

Conformance is defined separately for the core language and each of the Specialized Needs Annexes (see 1.1.3).

The following Annexes are informative:

- Annex K, “Language-Defined Aspects and Attributes”
- Annex L, “Language-Defined Pragmas”
- Annex M, “Summary of Documentation Requirements”
- Annex N, “Glossary”
- Annex P, “Syntax Summary”
- Annex Q, “Language-Defined Entities”

Each clause is divided into subclauses that have a common structure. Each clause and subclause first introduces its subject. After the introductory text, text is labeled with the following headings:

Syntax

Syntax rules (indented).

Name Resolution Rules

Compile-time rules that are used in name resolution, including overload resolution.

Legality Rules

Rules that are enforced at compile time. A construct is *legal* if it obeys all of the Legality Rules.

Static Semantics

A definition of the compile-time effect of each construct.

Post-Compilation Rules

Rules that are enforced before running a partition. A partition is legal if its compilation units are legal and it obeys all of the Post-Compilation Rules.

Dynamic Semantics

A definition of the run-time effect of each construct. 30

Bounded (Run-Time) Errors

Situations that result in bounded (run-time) errors (see 1.1.5). 31

Erroneous Execution

Situations that result in erroneous execution (see 1.1.5). 32

Implementation Requirements

Additional requirements for conforming implementations. 33

Documentation Requirements

Documentation requirements for conforming implementations. 34

Metrics

Documentation requirements for metrics of the time/space properties of the execution of certain language constructs. 35/6

Implementation Permissions

Additional permissions given to the implementer. 36

Implementation Advice

Optional advice given to the implementer. The word “should” is used to indicate that the advice is a recommendation, not a requirement. It is implementation defined whether or not a given recommendation is obeyed. 37

Usage

Advice and notes on the usage of the Ada programming language. This material is informative. 38/6

NOTE Notes emphasize consequences of the rules described in the (sub)clause or elsewhere. This material is informative. 39

Examples

Examples illustrate the possible forms of the constructs described. This material is informative. 40

1.1.3 Conformity of an Implementation

Implementation Requirements

A conforming implementation shall: 1

- Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation; 2
- Identify all programs or program units that are so large as to exceed the capacity of the implementation (or raise an appropriate exception at run time); 3
- Identify all programs or program units that contain errors whose detection is required by this Reference Manual; 4
- Supply all language-defined library units required by this Reference Manual; 5

6 - Contain no variations except those explicitly permitted by this Reference Manual, or those that are impossible or impractical to avoid given the implementation's execution environment;
7 - Specify all such variations in the manner prescribed by this Reference Manual.

8 The *external effect* of the execution of an Ada program is defined in terms of its interactions with its external environment. The following are defined as *external interactions*:

9 - Any interaction with an external file (see A.7);
10 - The execution of certain code_statements (see 13.8); which code_statements cause external interactions is implementation defined.
11 - Any call on an imported subprogram (see Annex B), including any parameters passed to it;
12 - Any result returned or exception propagated from a main subprogram (see 10.2) or an exported subprogram (see Annex B) to an external caller;
13 - Any read or update of an atomic or volatile object (see C.6);
14 - The values of imported and exported objects (see Annex B) at the time of any other interaction with the external environment.

15 A conforming implementation of this Reference Manual shall produce for the execution of a given Ada program a set of interactions with the external environment whose order and timing are consistent with the definitions and requirements of this Reference Manual for the semantics of the given program.

16/6 An implementation that fully conforms to the *core* of this Reference Manual shall support each capability required by the core language (see 1.1.2), as specified. Full conformance to a Specialized Needs Annex means that each capability required by the Annex shall be provided as specified.

17/6 An implementation conforming, either in full or in part, to this Reference Manual may provide additional aspects, attributes, library units, and pragmas. However, it shall not provide any aspect, attribute, library unit, or pragma having the same name as an aspect, attribute, library unit, or pragma (respectively) specified in the core or a Specialized Needs Annex unless the provided construct is either as specified in this Reference Manual or is more limited in capability than that required by this Reference Manual. A program that attempts to use an unsupported capability shall either be identified by the implementation before run time or shall raise an exception at run time.

17.1/6 A conforming implementation of a language-defined unit shall abide by all postconditions, type invariants, and default initial conditions specified for the unit by this document (see 11.4.2).

Documentation Requirements

18 Certain aspects of the semantics are defined to be either *implementation defined* or *unspecified*. In such cases, the set of possible effects is specified, and the implementation may choose any effect in the set. Implementations shall document their behavior in implementation-defined situations, but documentation is not required for unspecified situations. The implementation-defined characteristics are summarized in M.2.

19 The implementation may choose to document implementation-defined behavior either by documenting what happens in general, or by providing some mechanism for the user to determine what happens in a particular case.

Implementation Advice

20 If an implementation detects the use of an unsupported Specialized Needs Annex feature at run time, it should raise Program_Error if feasible.

If an implementation wishes to provide implementation-defined extensions to the functionality of a language-defined library unit, it should normally do so by adding children to the library unit. 21

NOTE The above requirements imply that an implementation conforming to this Reference Manual can support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities. 22/5

1.1.4 Method of Description and Syntax Notation

The form of an Ada program is described by means of a context-free syntax together with context-dependent requirements expressed by narrative rules. 1

The meaning of Ada programs is described by means of narrative rules defining both the effects of each construct and the composition rules for constructs. 2

The context-free syntax of the language is described using a simple variant of Backus-Naur Form. In particular: 3

- Lower case words in a sans-serif font, some containing embedded underlines, are used to denote syntactic categories, for example: 4

  case_statement 5

- Boldface words are used to denote reserved words, for example: 6

  array 7

- Square brackets enclose optional items. Thus the two following rules are equivalent. 8

  simple_return_statement ::= **return** [expression]; 9/2
  simple_return_statement ::= **return**; | **return** expression;

- Curly brackets enclose a repeated item. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus the two following rules are equivalent. 10

  term ::= factor {multiplying_operator factor} 11
  term ::= factor | term multiplying_operator factor

- A vertical line separates alternative items, for example: 12/5

  constraint ::= scalar_constraint | composite_constraint 13/5

- For symbols used in this notation (square brackets, curly brackets, and the vertical line), the symbols when surrounded by ' represent themselves, for example: 13.1/5

  discrete_choice_list ::= discrete_choice {'|' discrete_choice} 13.2/5
  named_container_aggregate ::= '[' container_element_association_list ']'

- If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example *subtype*_name and *task*_name are both equivalent to name alone. 14

The delimiters, compound delimiters, reserved words, and numeric_literals are exclusively made of the characters whose code point is between 16#20# and 16#7E#, inclusively. The special characters for which names are defined in this document (see 2.1) belong to the same range. For example, the character E in the definition of exponent is the character whose name is “LATIN CAPITAL LETTER E”, not “GREEK CAPITAL LETTER EPSILON”. 14.1/3

14.2/6 When this document mentions the conversion of some character or sequence of characters to upper case, it means the character or sequence of characters obtained by using simple upper case mapping, as defined by the Unicode Character Database.

15 A *syntactic category* is a nonterminal in the grammar defined in BNF under "Syntax". Names of syntactic categories are set in a different font, like_this.

16 A *construct* is a piece of text (explicit or implicit) that is an instance of a syntactic category defined under "Syntax".

17 A *constituent* of a construct is the construct itself, or any construct appearing within it.

18 Whenever the run-time semantics defines certain actions to happen in an *arbitrary order*, this means that the implementation shall arrange for these actions to occur in a way that is equivalent to some sequential order, following the rules that result from that sequential order. When evaluations are defined to happen in an arbitrary order, with conversion of the results to some subtypes, or with some runtime checks, the evaluations, conversions, and checks may be arbitrarily interspersed, so long as each expression is evaluated before converting or checking its value. Note that the effect of a program can depend on the order chosen by the implementation. This can happen, for example, if two actual parameters of a given call have side effects.

19 NOTE 1 The syntax rules describing structured constructs are presented in a form that corresponds to the recommended paragraphing. For example, an if_statement is defined as:

20
```
if_statement ::=
     if condition then
        sequence_of_statements
    {elsif condition then
        sequence_of_statements}
    [else
        sequence_of_statements]
     end if;
```

21 NOTE 2 The line breaks and indentation in the syntax rules indicate the recommended line breaks and indentation in the corresponding constructs. The preferred places for other line breaks are after semicolons.

1.1.5 Classification of Errors

Implementation Requirements

1 The language definition classifies errors into several different categories:

2 • Errors that are required to be detected prior to run time by every Ada implementation;

3 These errors correspond to any violation of a rule given in this document, other than those listed below. In particular, violation of any rule that uses the terms shall, allowed, permitted, legal, or illegal belongs to this category. Any program that contains such an error is not a legal Ada program; on the other hand, the fact that a program is legal does not mean, *per se*, that the program is free from other forms of error.

4 The rules are further classified as either compile time rules, or post compilation rules, depending on whether a violation has to be detected at the time a compilation unit is submitted to the compiler, or may be postponed until the time a compilation unit is incorporated into a partition of a program.

5 • Errors that are required to be detected at run time by the execution of an Ada program;

6 The corresponding error situations are associated with the names of the predefined exceptions. Every Ada compiler is required to generate code that raises the corresponding exception if such an error situation arises during program execution. If such an error situation is certain to arise in

every execution of a construct, then an implementation is allowed (although not required) to report this fact at compilation time.

- Bounded errors; 7

  The language rules define certain kinds of errors that are not expected to be detected either prior to or during run time, but if not detected, the range of possible effects shall be bounded. The errors of this category are called *bounded errors*. The possible effects of a given bounded error are specified for each such error, but in any case one possible effect of a bounded error is the raising of the exception Program_Error. 8/5

- Erroneous execution. 9

  In addition to bounded errors, the language rules define certain kinds of errors as leading to *erroneous execution*. Like bounded errors, the implementation is not expected to detect such errors either prior to or during run time. Unlike bounded errors, there is no language-specified bound on the possible effect of erroneous execution; the effect is in general not predictable. 10/5

Implementation Permissions

An implementation may provide *nonstandard modes* of operation. Typically these modes would be selected by a pragma or by a command line switch when the compiler is invoked. When operating in a nonstandard mode, the implementation may reject compilation_units that do not conform to additional requirements associated with the mode, such as an excessive number of warnings or violation of coding style guidelines. Similarly, in a nonstandard mode, the implementation may apply special optimizations or alternative algorithms that are only meaningful for programs that satisfy certain criteria specified by the implementation. In any case, an implementation shall support a *standard* mode that conforms to the requirements of this Reference Manual; in particular, in the standard mode, all legal compilation_units shall be accepted. 11

Implementation Advice

If an implementation detects a bounded error or erroneous execution, it should raise Program_Error. 12

1.2 Normative References

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies. For other documents mentioned in this document, see 1.2.1, “Bibliography”. 1/3

Paragraphs 1 through 8 have been moved to the Bibliography.

ISO 639-3:2007, *Codes for the representation of names of languages — Part 3: Alpha-3 code for comprehensive coverage of languages* 1.1/3

ISO 3166-1:2020, *Codes for the representation of names of countries and their subdivisions — Part 1: Country codes* 4.1/5

8/6

The Unicode Consortium. The Unicode Standard, Version 15.0.0, (Mountain View, CA: The Unicode Consortium, 2022. ISBN 978-1-936213-32-0) Available at: https://www.unicode.org/versions/Unicode15.0.0/ 9/6

10/6 Unicode Standard Annex #44, "Unicode Character Database", edited by Ken Whistler, an integral part of The Unicode Standard. Available at: https://www.unicode.org/reports/tr44/

1.2.1 Bibliography

1/5 The following documents are mentioned in this document as informative references.

2/5 ISO/IEC 646:1991, *Information technology — ISO 7-bit coded character set for information interchange*

3/5 ISO/IEC 1539-1:2018, *Information technology — Programming languages — Fortran — Part 1: Base language*

4/5 ISO/IEC 1989:2002, *Information technology — Programming languages — COBOL*

5/5 ISO/IEC 6429:1992, *Information technology — Control functions for coded character sets*

6/5 ISO 8601-1:2019, *Date and time — Representations for information interchange — Part 1: Basic rules*

7/5 ISO/IEC 8859-1:1998, *Information technology — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*

8/5 ISO/IEC 9899:2018, *Information technology — Programming languages — C*

8.1/6 ISO/IEC 10646:2020, *Information technology — Universal coded character set (UCS)*

9/5 ISO/IEC 14882:2020, *Programming languages — C++*

10/5 ISO/IEC 60559:2020, *Information technology — Microprocessor Systems — Floating-Point arithmetic*

1.3 Terms and Definitions

1/5 Terms are defined throughout this document, indicated by *italic* type. Terms explicitly defined in this document are not to be presumed to refer implicitly to similar terms defined elsewhere. Mathematical terms not defined in this document are to be interpreted according to the *CRC Concise Encyclopedia of Mathematics, Second Edition*. Other terms not defined in this document are to be interpreted according to the *Webster's Third New International Dictionary of the English Language*. Informal descriptions of some terms are also given below.

1.3.1 Types, Objects, and their Properties

1/5 **abstract type**
tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own

2/5 **access type**
type that has values that designate aliased objects

3/5 Note: Access types correspond to “pointer types” or “reference types” in some other languages.

4/5 **accessibility level**
representation of the lifetime of an entity in terms of the level of dynamic nesting within which the entity is known to exist

5/5 **aliased view**
view of an object that can be designated by an access value

Note: Objects allocated by allocators are aliased. Objects can also be explicitly declared as aliased with the reserved word aliased. The Access attribute can be used to create an access value designating an aliased object. 6/5

ancestor of a type 7/5
type itself or, in the case of a type derived from other types, its parent type or one of its progenitor types or one of their ancestors

Note: Ancestor and descendant are inverse relationships. 8/5

array type 9/5
composite type whose components are all of the same type

aspect 10/5
specifiable property of an entity

Note: An aspect can be specified by an aspect_specification on the declaration of the entity. Some aspects can be queried via attributes. 11/5

attribute 12/5
characteristic or property of an entity that can be queried, and in some cases specified

category of types 13/5
set of types with one or more common properties, such as primitive operations

Note: A category of types that is closed under derivation is also known as a class. 14/5

character type 15/5
enumeration type whose values include characters

class of types 16/5
set of types that is closed under derivation, which means that if a given type is in the class, then all types derived from that type are also in the class

Note: The set of types of a class share common properties, such as their primitive operations. 17/5

composite type 18/5
type with components, such as an array or record

controlled type 19/5
type that supports user-defined assignment and finalization

Note: Objects are always finalized before being destroyed. 20/5

default initial condition 21/5
property that holds for every default-initialized object of a given type

derived type 22/5
type defined in terms of a parent type and zero or more progenitor types given in a derived type definition

Note 1: A derived type inherits properties such as components and primitive operations from its parent and progenitors. 23/5

Note 2: A type together with the types derived from it (directly or indirectly) form a derivation class. 24/5

descendant of a type 25/5
type itself or a type derived (directly or indirectly) from it

Note: Descendant and ancestor are inverse relationships. 26/5

discrete type 27/5
type that is either an integer type or an enumeration type

discriminant 28/5
parameter for a composite type, which can control, for example, the bounds of a component that is an array

29/5 Note: A discriminant for a task type can be used to pass data to a task of the type upon its creation.

30/5 **elementary type**
type that does not have components

31/5 **enumeration type**
type defined by an enumeration of its values, which can be denoted by identifiers or character literals

32/5 **full type**
type that defines a full view

33/5 **full view**
view of a type that reveals all of its properties

34/5 Note: There can be other views of the type that reveal fewer properties.

35/5 **incomplete type**
type that defines an incomplete view

36/5 Note: Incomplete types can be used for defining recursive data structures.

37/5 **incomplete view**
view of a type that reveals minimal properties

38/5 Note: The remaining properties are defined by the full view given elsewhere.

39/5 **indexable container type**
type that has user-defined behavior for indexing, via the Constant_Indexing or Variable_Indexing aspects

40/5 **integer type**
type that represents signed or modular integers

41/5 Note: A signed integer type has a base range that includes both positive and negative numbers, and has operations that can raise an exception when the result is outside the base range. A modular type has a base range whose lower bound is zero, and has operations with "wraparound" semantics. Modular types subsume what are called "unsigned types" in some other languages.

42/5 **interface type**
abstract tagged type that has no components or concrete operations except possibly null procedures

43/5 Note: Interface types are used for composing other interfaces and tagged types and thereby provide multiple inheritance. Only an interface type can be used as a progenitor of another type.

44/5 **invariant**
assertion that is expected to be True for all objects of a given private type when viewed from outside the defining package

45/5 **iterable container type**
type that has user-defined behavior for iteration, via the Default_Iterator and Iterator_Element aspects

46/5 **limited type**
type for which copying (such as in an assignment_statement) is not allowed

47/5 Note: All types are either limited types or nonlimited types.

48/5 **needed component**
component of a record type or record extension that is required to have its value specified within a given aggregate

49/5 **nominal subtype**
subtype specified when a view of an object is defined

50/5 **nonlimited type**
type for which copying is allowed

object 51/5
entity that contains a value, and is either a constant or a variable

Note: An object is created by an object_declaration or by an allocator. A formal parameter is (a view of) an object. A subcomponent of an object is an object. 52/5

operational aspect 53/5
aspect that indicates a logical property of an entity, such as the precondition of a subprogram, or the procedure used to write a given type of object to a stream

parent of a derived type 54/5
first ancestor type given in the definition of the derived type

Note: The parent can be almost any kind of type, including an interface type. 55/5

partial view 56/5
view of a type that reveals only some of its properties

Note: The remaining properties are defined by the full view given elsewhere. 57/5

primitive operations of a type 58/5
operations (such as subprograms) declared together with the type declarations

Note: Primitive operations are inherited by other types in the same derivation class of types. 59/5

private extension 60/5
type that extends another type, with the additional properties hidden from its clients

private type 61/5
type that defines a partial view

Note: Private types can be used for defining abstractions that hide unnecessary details from their clients. 62/5

progenitor 63/5
type given in the interface list, if any, of an interface, task, protected, or derived type definition

Note: A progenitor is always an interface type. 64/5

protected type 65/5
composite type whose components are accessible only through one of its protected operations, which synchronize concurrent access by multiple tasks

real type 66/5
type that has values that are approximations of the real numbers

Note: Floating point and fixed point types are real types. 67/5

record extension 68/5
type that extends another type optionally with additional components

record type 69/5
composite type consisting of zero or more named components, possibly of different types

reference type 70/5
type that has user-defined behavior for ".all", defined by the Implicit_Dereference aspect

representation aspect 71/5
aspect that indicates how an entity is mapped onto the underlying hardware, for example the size or alignment of an object

scalar type 72/5
either a discrete type or a real type

stable property 73/5
characteristic associated with objects of a given type that is preserved by many of the primitive operations of the type

74/5 **storage pool object**
object associated with one or more access types from which the storage for objects created by allocators of the access type(s) is obtained

75/5 Note: Some storage pools can be partitioned into subpools in order to support finer-grained storage management.

76/5 **stream**
sequence of elements that can be used, along with the stream-oriented attributes, to support marshalling and unmarshalling of values of most types

77/5 **subtype**
type together with optional constraints, null exclusions, and predicates, which constrain the values of the type to the subset that satisfies the implied conditions

78/5 **synchronized**
can be safely operated on by multiple tasks concurrently

79/5 Note: Synchronized is used to qualify entities, as in a synchronized interface.

80/5 **tagged type**
type whose objects each have a run-time type tag, which indicates the specific type for which the object was originally created

81/5 Note: Tagged types can be extended with additional components.

82/5 **task type**
composite type used to represent active entities which execute concurrently and that can communicate via queued task entries

83/5 Note: The top-level task of a partition is called the environment task.

84/5 **type**
defining characteristic of each object and expression of the language, with an associated set of values, and a set of primitive operations that implement the fundamental aspects of its semantics

85/5 Note: Types are grouped into categories. Most language-defined categories of types are also classes of types.

86/5 **view of an entity**
representation of an entity that reveals some or all of the properties of the entity

87/5 Note: A single entity can have multiple views.

1.3.2 Subprograms and their Properties

1/5 **function**
form of subprogram that returns a result and can be called as part of an expression

2/5 **overriding operation**
operation that replaces an inherited primitive operation

3/5 Note: Operations can be marked explicitly as overriding or not overriding.

4/5 **postcondition**
assertion that is expected to be True when a given subprogram returns normally

5/5 **precondition**
assertion that is expected to be True when a given subprogram is called

6/5 **procedure**
form of subprogram that does not return a result and can only be invoked by a statement

7/5 **subprogram**
unit of a program that can be brought into execution in various contexts, with the invocation being a subprogram call that can parameterize the effect of the subprogram through the passing of operands

Note: There are two forms of subprograms: functions, which return values, and procedures, which do not. 8/5

1.3.3 Other Syntactic Constructs

aggregate 1/5
construct used to define a value of a composite type by specifying the values of the components of the type

compilation unit 2/5
program unit that is separately compiled

Note: A compilation_unit contains either the declaration, the body, or a renaming of a program unit. 3/5

construct 4/5
piece of text (explicit or implicit) that is an instance of a syntactic category defined under Syntax

container 5/5
structured object that represents a collection of elements all of the same (potentially class-wide) type, such as a vector or a tree

Note: Several predefined container types are provided by the children of package Ada.Containers (see A.18.1). 6/5

container aggregate 7/5
construct used to define a value of a type that represents a collection of elements, by explicitly specifying the elements in the collection

core language 8/5
clause or annex in which are defined language constructs or capabilities that are provided by all conforming implementations

Note: A construct is said to be part of the core language if it is defined in a core language clause or annex. 9/5

declaration 10/5
language construct that associates a name with (a view of) an entity

Note: A declaration can appear explicitly in the program text (an explicit declaration), or can be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an implicit declaration). 11/5

generic instance 12/5
nongeneric unit created by the instantiation of a generic unit

generic unit 13/5
template for a (nongeneric) program unit

Note 1: The template can be parameterized by objects, types, subprograms, and packages. 14/5

Note 2: Generic units can be used to perform the role that macros sometimes play in other languages. 15/5

iterator 16/5
construct that is used to loop over the elements of an array or container

Note: Iterators can be user defined, and can perform arbitrary computations to access elements from a container. 17/5

iterator filter 18/5
construct that is used to restrict the elements produced by an iteration to those for which a boolean condition evaluates to True

library unit 19/5
separately compiled program unit, which is a package, a subprogram, or a generic unit

Note: Library units can have other (logically nested) library units as children, and can have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a subsystem. 20/5

21/5 **master construct**
one of certain executable constructs for which there can be objects or tasks whose lifetime ends when the construct completes

22/5 Note: Execution of a master construct is a master, with which objects and tasks are associated for the purposes of waiting and finalization.

23/5 **needed compilation unit**
compilation unit that is necessary to produce an executable partition, because some entity declared or defined within the unit is used elsewhere in the partition

24/5 **package**
program unit that defines the interface to a group of logically related entities, along with their implementation

25/5 Note: Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users.

26/5 **parallel construct**
executable construct that defines multiple activities of a single task that can proceed in parallel, via the execution of multiple logical threads of control

27/5 **partition**
part of a program, which consists of a set of interdependent library units

28/5 Note: Each partition can run in a separate address space, possibly on a separate computer. A program can contain just one partition, or it can be distributed across multiple partitions, which can execute concurrently.

29/5 **pragma**
compiler directive to provide control over and above that provided by the other syntactic constructs of the language

30/5 Note: There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation can support additional (implementation-defined) pragmas.

31/5 **program**
set of partitions, each of which can execute in a separate address space, possibly on a separate computer

32/5 **program unit**
language construct that is a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal

33/5 Note: Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units.

34/5 **reduction expression**
expression that defines how to map or transform a collection of values into a new set of values, and then summarize the values by applying an operation to reduce the set to a single value

35/5 **renaming**
declaration that does not define a new entity, but instead defines a new view of an existing entity

36/5 **specialized needs annex**
annex in which are defined language constructs or capabilities that are not necessarily provided by all conforming implementations

37/5 **subunit**
body of a program unit that can be compiled separately from its enclosing program unit

1.3.4 Runtime Actions

assertion 1/5
boolean expression that is expected to be True at run time at certain specified places

Note: Certain pragmas and aspects define various kinds of assertions. 2/5

elaboration 3/5
process by which a declaration achieves its run-time effect

Note: Elaboration is one of the forms of execution. 4/5

evaluation 5/5
process by which an expression achieves its run-time effect

Note: Evaluation is one of the forms of execution. 6/5

execution 7/5
process by which a construct achieves its run-time effect

Note: Execution of a declaration is also called elaboration. Execution of an expression is also called evaluation. 8/5

logical thread of control 9/5
activity within the execution of a program that can proceed in parallel with other activities of the same task, or of separate tasks

master 10/5
execution of a master construct

Note: Each object and task is associated with a master. When a master is left, associated tasks are awaited and associated objects are finalized. 11/5

1.3.5 Exceptional Situations

check 1/5
test made during execution to determine whether a language rule has been violated

exception 2/5
kind of exceptional situation

exception occurrence 3/5
run-time occurrence of an exceptional situation

handle an exception 4/5
perform some actions in response to the arising of an exception

raise an exception 5/5
abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen

suppress a check 6/5
assert that the check cannot fail, and request that the compiler optimize by disabling the check

Note: The compiler is not required to honor this request. Suppressing checks that can fail can cause a program to behave in arbitrary ways. 7/5

2 Lexical Elements

The text of a program consists of the texts of one or more compilations. The text of a compilation is a sequence of lexical elements, each composed of characters; the rules of composition are given in this clause. Pragmas, which provide certain information for the compiler, are also described in this clause. 1/3

2.1 Character Set

The character repertoire for the text of an Ada program consists of the entire codespace described by The Unicode Standard, version 15.0.0. This codespace is organized in *planes*, each plane comprising 65536 characters. 1/6

Syntax

Paragraphs 2 and 3 were deleted.

A character is defined by this Reference Manual for each code point in the codespace described by The Unicode Standard, version 15.0.0, regardless of whether or not Unicode allocates a character to that code point. 3.1/6

Static Semantics

The coded representation for characters is implementation defined (it can be a representation that is not defined within The Unicode Standard, version 15.0.0). A character whose relative code point in its plane is 16#FFFE# or 16#FFFF# is not allowed anywhere in the text of a program. The only characters allowed outside of comments are those in categories other_format, format_effector, and graphic_character. 4/6

The semantics of an Ada program whose text is not in Normalization Form C (as defined by Definition D120 of The Unicode Standard, version 15.0.0) is implementation defined. 4.1/6

The description of the language definition in this document uses the character properties General Category, Simple Uppercase Mapping, and Simple Case Folding of the Unicode Character Database. The values of the General Category are referenced by their description in Table 4-4 of the Unicode Standard. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified. 5/6

Characters are categorized as follows: 6/3

This paragraph was deleted. 7/2

letter_uppercase 8/2
: Any character whose General Category is defined to be “Letter, Uppercase”.

letter_lowercase 9/2
: Any character whose General Category is defined to be “Letter, Lowercase”.

letter_titlecase 9.1/2
: Any character whose General Category is defined to be “Letter, Titlecase”.

letter_modifier 9.2/2
: Any character whose General Category is defined to be “Letter, Modifier”.

letter_other 9.3/2
: Any character whose General Category is defined to be “Letter, Other”.

mark_non_spacing 9.4/2
: Any character whose General Category is defined to be “Mark, Non-Spacing”.

9.5/2 mark_spacing_combining
Any character whose General Category is defined to be “Mark, Spacing Combining”.

10/2 number_decimal
Any character whose General Category is defined to be “Number, Decimal”.

10.1/2 number_letter
Any character whose General Category is defined to be “Number, Letter”.

10.2/2 punctuation_connector
Any character whose General Category is defined to be “Punctuation, Connector”.

10.3/2 other_format
Any character whose General Category is defined to be “Other, Format”.

11/2 separator_space
Any character whose General Category is defined to be “Separator, Space”.

12/2 separator_line
Any character whose General Category is defined to be “Separator, Line”.

12.1/2 separator_paragraph
Any character whose General Category is defined to be “Separator, Paragraph”.

13/3 format_effector
The characters whose code points are 16#09# (CHARACTER TABULATION), 16#0A# (LINE FEED), 16#0B# (LINE TABULATION), 16#0C# (FORM FEED), 16#0D# (CARRIAGE RETURN), 16#85# (NEXT LINE), and the characters in categories separator_line and separator_paragraph.

13.1/2 other_control
Any character whose General Category is defined to be “Other, Control”, and which is not defined to be a format_effector.

13.2/2 other_private_use
Any character whose General Category is defined to be “Other, Private Use”.

13.3/2 other_surrogate
Any character whose General Category is defined to be “Other, Surrogate”.

14/3 graphic_character
Any character that is not in the categories other_control, other_private_use, other_surrogate, format_effector, and whose relative code point in its plane is neither 16#FFFE# nor 16#FFFF#.

The following names are used when referring to certain characters (the first name is that given in The Unicode Standard, version 15.0.0): 15/6

graphic symbol	name	graphic symbol	name
"	quotation mark	:	colon
#	number sign	;	semicolon
&	ampersand	<	less-than sign
'	apostrophe, tick	=	equals sign
(	left parenthesis	>	greater-than sign
)	right parenthesis	_	low line, underline
*	asterisk, multiply	|	vertical line
+	plus sign	/	solidus, divide
,	comma	!	exclamation point
–	hyphen-minus, minus	%	percent sign
.	full stop, dot, point	[	left square bracket
@	commercial at, at sign	]	right square bracket

Implementation Requirements

An Ada implementation shall accept Ada source code in UTF-8 encoding, with or without a BOM (see A.4.11), where every character is represented by its code point. The character pair CARRIAGE RETURN/LINE FEED (code points 16#0D# 16#0A#) signifies a single end of line (see 2.2); every other occurrence of a format_effector other than the character whose code point position is 16#09# (CHARACTER TABULATION) also signifies a single end of line. 16/3

Implementation Permissions

The categories defined above, as well as case mapping and folding, may be based on an implementation-defined version of the Unicode Character Database (4.0 or later). 17/6

NOTE The characters in categories other_control, other_private_use, and other_surrogate are only allowed in comments. 18/2

2.2 Lexical Elements, Separators, and Delimiters

Static Semantics

The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate *lexical elements*. Each lexical element is formed from a sequence of characters, and is either a delimiter, an identifier, a reserved word, a numeric_literal, a character_literal, a string_literal, or a comment. The meaning of a program depends only on the particular sequences of lexical elements that form its compilations, excluding comments. 1

The text of a compilation is divided into *lines*. In general, the representation for an end of line is implementation defined. However, a sequence of one or more format_effectors other than the character whose code point is 16#09# (CHARACTER TABULATION) signifies at least one end of line. 2/3

In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a separator_space, a format_effector, or the end of a line, as follows: 3/2

- A separator_space is a separator except within a comment, a string_literal, or a character_literal. 4/2

5/3 • The character whose code point is 16#09# (CHARACTER TABULATION) is a separator except within a comment.

6 • The end of a line is always a separator.

7 One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric_literal and an adjacent identifier, reserved word, or numeric_literal.

7.1/3 One or more other_format characters are allowed anywhere that a separator is; any such characters have no effect on the meaning of an Ada program.

8/2 A *delimiter* is either one of the following characters:

9/5 & ' ( ) * + , – . / : ; < = > @ [ ] |

10 or one of the following *compound delimiters* each composed of two adjacent special characters

11 => .. ** := /= >= <= << >> <>

12 Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string_literal, character_literal, or numeric_literal.

13 The following names are used when referring to compound delimiters:

delimiter	name
=>	arrow
..	double dot
**	double star, exponentiate
:=	assignment (pronounced: “becomes”)
/=	inequality (pronounced: “not equal”)
>=	greater than or equal
<=	less than or equal
<<	left label bracket
>>	right label bracket
<>	box

Implementation Requirements

14 An implementation shall support lines of at least 200 characters in length, not counting any characters used to signify the end of a line. An implementation shall support lexical elements of at least 200 characters in length. The maximum supported line length and lexical element length are implementation defined.

2.3 Identifiers

Identifiers are used as names. 1

Syntax

identifier ::= 2/2
  identifier_start {identifier_start | identifier_extend}

identifier_start ::= 3/2
    letter_uppercase
  | letter_lowercase
  | letter_titlecase
  | letter_modifier
  | letter_other
  | number_letter

identifier_extend ::= 3.1/3
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector

An identifier shall not contain two consecutive characters in category punctuation_connector, or end with a character in that category. 4/3

Legality Rules

An identifier shall only contain characters that may be present in Normalization Form KC as defined by Definition D121 of The Unicode Standard, version 15.0.0. 4.1/6

Static Semantics

Two identifiers are considered the same if they consist of the same sequence of characters after applying simple case folding, as defined by Section 3.13 of Unicode Version 15.0. 5/6

After applying simple case folding, an identifier shall not be identical to a reserved word. 5.3/3

Implementation Permissions

In a nonstandard mode, an implementation may support other upper/lower case equivalence rules for identifiers, to accommodate local conventions. 6

NOTE Identifiers differing only in the use of corresponding upper and lower case letters are considered the same. 6.1/2

Examples

Examples of identifiers: 7

```
Count       X     Get_Symbol    Ethelyn   Marion                    8/2
Snobol_4    X1    Page_Count    Store_Next_Item
Πλάτων        -- Plato
Чайковский    -- Tchaikovsky
θ  φ          -- Angles
```

2.4 Numeric Literals

1 There are two kinds of numeric_literals, *real literals* and *integer literals*. A real literal is a numeric_literal that includes a point; an integer literal is a numeric_literal without a point.

Syntax

2 numeric_literal ::= decimal_literal | based_literal

3 NOTE The type of an integer literal is *universal_integer*. The type of a real literal is *universal_real*.

2.4.1 Decimal Literals

1 A decimal_literal is a numeric_literal in the conventional decimal notation (that is, the base is ten).

Syntax

2 decimal_literal ::= numeral [.numeral] [exponent]

3 numeral ::= digit {[underline] digit}

4 exponent ::= E [+] numeral | E – numeral

4.1/2 digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

5 An exponent for an integer literal shall not have a minus sign.

Static Semantics

6 An underline character in a numeric_literal does not affect its meaning. The letter E of an exponent can be written either in lower case or in upper case, with the same meaning.

7 An exponent indicates the power of ten by which the value of the decimal_literal without the exponent is to be multiplied to obtain the value of the decimal_literal with the exponent.

Examples

8 *Examples of decimal literals:*

9
```
12          0       1E6     123_456     -- integer literals
12.0        0.0     0.456   3.14159_26  -- real literals
```

2.4.2 Based Literals

1 A based_literal is a numeric_literal expressed in a form that specifies the base explicitly.

Syntax

2 based_literal ::=
  base # based_numeral [.based_numeral] # [exponent]

3 base ::= numeral

4 based_numeral ::=
  extended_digit {[underline] extended_digit}

5 extended_digit ::= digit | A | B | C | D | E | F

Legality Rules

The *base* (the numeric value of the decimal numeral preceding the first #) shall be at least two and at most sixteen. The extended_digits A through F represent the digits ten through fifteen, respectively. The value of each extended_digit of a based_literal shall be less than the base. 6

Static Semantics

The conventional meaning of based notation is assumed. An exponent indicates the power of the base by which the value of the based_literal without the exponent is to be multiplied to obtain the value of the based_literal with the exponent. The base and the exponent, if any, are in decimal notation. 7

The extended_digits A through F can be written either in lower case or in upper case, with the same meaning. 8

Examples

Examples of based literals: 9

```
2#1111_1111#  16#FF#      016#0ff#   -- integer literals of value 255
16#E#E1       2#1110_0000#           -- integer literals of value 224
16#F.FF#E+2   2#1.1111_1111_1110#E11 -- real literals of value 4095.0
```
10

2.5 Character Literals

A character_literal is formed by enclosing a graphic character between two apostrophe characters. 1

Syntax

character_literal ::= 'graphic_character' 2

NOTE A character_literal is an enumeration literal of a character type. See 3.5.2. 3

Examples

Examples of character literals: 4

```
'A'      '*'      '''      ' '
'L'      'Л'      'Λ'      -- Various els.
'∞'      'ℵ'               -- Big numbers - infinity and aleph.
```
5/2

2.6 String Literals

A string_literal is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks used as string brackets. They are used to represent operator_symbols (see 6.1), values of a string type (see 4.2), and array subaggregates (see 4.3.3). 1

Syntax

string_literal ::= "{string_element}" 2

string_element ::= "" | *non_quotation_mark*_graphic_character 3

A string_element is either a pair of quotation marks (""), or a single graphic_character other than a quotation mark. 4

Static Semantics

The *sequence of characters* of a string_literal is formed from the sequence of string_elements between the bracketing quotation marks, in the given order, with a string_element that is "" becoming a single 5

quotation mark in the sequence of characters, and any other string_element being reproduced in the sequence.

6 A *null string literal* is a string_literal with no string_elements between the quotation marks.

7 NOTE 1 An end of line cannot appear in a string_literal.

7.1/2 NOTE 2 No transformation is performed on the sequence of characters of a string_literal.

Examples

8 *Examples of string literals:*

9/2
```
"Message of the day:"

""                      -- a null string literal
" "   "A"   """"        -- three string literals of length 1

"Characters such as $, %, and } are allowed in string literals"
"Archimedes said ""Εύρηκα"""
"Volume of cylinder (πr²h) = "
```

2.7 Comments

1 A comment starts with two adjacent hyphens and extends up to the end of the line.

Syntax

2 comment ::= --{*non_end_of_line*_character}

3 A comment may appear on any line of a program.

Static Semantics

4 The presence or absence of comments has no influence on whether a program is legal or illegal. Furthermore, comments do not influence the meaning of a program; their sole purpose is the enlightenment of the human reader.

Examples

5 *Examples of comments:*

6/5
```
-- the last sentence above echoes the Algol 68 report

end;  -- processing of Line is complete

-- a long comment can be split onto
-- two or more consecutive lines

---------------- the first two hyphens start the comment
```

2.8 Pragmas

1 A pragma is a compiler directive. There are language-defined pragmas that give instructions for optimization, listing control, etc. An implementation may support additional (implementation-defined) pragmas.

Syntax

2 pragma ::=
  pragma identifier [(pragma_argument_association {, pragma_argument_association})];

pragma_argument_association ::= 3/3
  [*pragma_argument_*identifier =>] name
| [*pragma_argument_*identifier =>] expression
| *pragma_argument_*aspect_mark => name
| *pragma_argument_*aspect_mark => expression

In a pragma, any pragma_argument_associations without a *pragma_argument_*identifier or *pragma_argument_*aspect_mark shall precede any associations with a *pragma_argument_*identifier or *pragma_argument_*aspect_mark. 4/3

There are five kinds of pragmas: configuration pragmas (and pragmas usable as configuration pragmas), context pragmas, aspect-related pragmas, executable pragmas, and control pragmas, and each has its own placement restrictions: 5/6

- A *configuration* pragma is allowed at the beginning of a compilation, preceded only by other pragmas. In addition, certain pragmas are *usable as configuration pragmas*, and can also control configuration more locally; as such these pragmas can appear immediately within a declarative_part or a package_specification at the place of a basic_declaration, as defined for specific pragmas. 6/6
- A *context* pragma is allowed at the place of a use_clause in a context_clause. 7/6
- An *aspect-related* pragma is allowed in place of an aspect_clause, or immediately following (except for other pragmas) a compilation unit. 7.1/6
- An *executable* pragma is allowed in place of a simple_statement, or in place of a basic_declaration or a declare_item. 7.2/6
- Unless specified otherwise, a *control* pragma is allowed at the following places in a program: 7.3/6
  - After a semicolon delimiter, but not within a formal_part or discriminant_part. 7.4/6
  - At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "item", "statement", "clause", or "alternative", or one of the syntactic categories variant or exception_handler; but not in place of such a construct if the construct is required, or is part of a list that is required to have at least one such construct. 7.5/6

Additional syntax rules and placement restrictions exist for specific pragmas. 8

The *name* of a pragma is the identifier following the reserved word **pragma**. The name or expression of a pragma_argument_association is a *pragma argument*. 9

An *identifier specific to a pragma* is an identifier or reserved word that is used in a pragma argument with special meaning for that pragma. 10/3

Static Semantics

If an implementation does not recognize the name of a pragma, then it has no effect on the semantics of the program. Inside such a pragma, the only rules that apply are the Syntax Rules. 11

Dynamic Semantics

An executable or aspect-related pragma that appears in place of an executable construct is executed, and is treated for the purposes of other rules of the language as being a simple_statement, basic_declarative_item, or declare_item according to where it appears. Unless otherwise specified for a particular pragma, this execution consists of the evaluation of each evaluable pragma argument in an arbitrary order. 12/6

Implementation Requirements

13 The implementation shall give a warning message for an unrecognized pragma name.

Implementation Permissions

14 An implementation may provide implementation-defined pragmas; the name of an implementation-defined pragma shall differ from those of the language-defined pragmas.

15 An implementation may ignore an unrecognized pragma even if it violates some of the Syntax Rules, if detecting the syntax error is too complex.

Implementation Advice

16/3 Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas in a working program are replaced with unrecognized pragmas, the program should still be legal, and should still have the same semantics.

17 Normally, an implementation should not define pragmas that can make an illegal program legal, except as follows:

18/3 • A pragma used to complete a declaration;

19 • A pragma used to configure the environment by adding, removing, or replacing library_items.

Syntax

20/6 The forms of the List, Page, and Optimize control pragmas are as follows:

21 **pragma** List(identifier);

22 **pragma** Page;

23 **pragma** Optimize(identifier);

24 Other pragmas are defined throughout this Reference Manual, and are summarized in Annex L.

Static Semantics

25 A pragma List takes one of the identifiers On or Off as the single argument. This pragma is allowed anywhere a pragma is allowed. It specifies that listing of the compilation is to be continued or suspended until a List pragma with the opposite argument is given within the same compilation. The pragma itself is always listed if the compiler is producing a listing.

26 A pragma Page is allowed anywhere a pragma is allowed. It specifies that the program text which follows the pragma should start on a new page (if the compiler is currently producing a listing).

27 A pragma Optimize takes one of the identifiers Time, Space, or Off as the single argument. This pragma is allowed anywhere a pragma is allowed, and it applies until the end of the immediately enclosing declarative region, or for a pragma at the place of a compilation_unit, to the end of the compilation. It gives advice to the implementation as to whether time or space is the primary optimization criterion, or that optional optimizations should be turned off. It is implementation defined how this advice is followed.

Examples

28 *Examples of pragmas:*

29/5
```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Assertion_Policy(Check); -- check assertions
pragma Assert(Exists(File_Name),
              Message => "Nonexistent file"); -- assert file exists
```

2.9 Reserved Words

Syntax

This paragraph was deleted. 1/1

The following are the *reserved words*. Within a program, some or all of the letters of a reserved word may be in upper case. 2/3

abort
abs
abstract
accept
access
aliased
all
and
array
at

begin
body

case
constant

declare
delay
delta
digits
do

else
elsif
end
entry
exception
exit

for
function

generic
goto

if
in
interface
is

limited
loop

mod

new
not
null

of
or
others
out
overriding

package
parallel
pragma
private
procedure
protected

raise
range
record
rem
renames
requeue

return
reverse

select
separate
some
subtype
synchronized

tagged
task
terminate
then
type

until
use

when
while
with

xor

NOTE The reserved words appear in **lower case boldface** in this document, except when used in the designator of an attribute (see 4.1.4). Lower case boldface is also used for a reserved word in a string_literal used as an operator_symbol. This is merely a convention — programs can be written in whatever typeface is desired and available. 3/5

3 Declarations and Types

This clause describes the types in the language and the rules for declaring constants, variables, and named numbers. 1/3

3.1 Declarations

The language defines several kinds of named *entities* that are declared by declarations. The entity's *name* is defined by the declaration, usually by a defining_identifier, but sometimes by a defining_character_literal or defining_operator_symbol. There are also entities that are not directly declared; some of these are elements of other entities, or are allocated dynamically. Such entities can be denoted using indexed_component, selected_component, or dereference names (see 4.1). 1/5

There are several forms of declaration. A basic_declaration is a form of declaration defined as follows. 2

Syntax

basic_declaration ::= 3/3

type_declaration	| subtype_declaration
| object_declaration	| number_declaration
| subprogram_declaration	| abstract_subprogram_declaration
| null_procedure_declaration	| expression_function_declaration
| package_declaration	| renaming_declaration
| exception_declaration	| generic_declaration
| generic_instantiation	

defining_identifier ::= identifier 4

Static Semantics

A *declaration* is a language construct that associates a name with (a view of) an entity. A declaration may appear explicitly in the program text (an *explicit* declaration), or may be supposed to occur at a given place in the text as a consequence of the semantics of another construct (an *implicit* declaration). 5

Each of the following is defined to be a declaration: any basic_declaration; an enumeration_literal_specification; a discriminant_specification; a component_declaration; a defining_identifier of an iterated_component_association; a loop_parameter_specification; a defining_identifier of a chunk_specification; an iterator_specification; a defining_identifier of an iterator_parameter_specification; a parameter_specification; a subprogram_body; an extended_return_object_declaration; an entry_declaration; an entry_index_specification; a choice_parameter_specification; a generic_formal_parameter_declaration. 6/5

All declarations contain a *definition* for a *view* of an entity. A view consists of an identification of the entity (the entity *of* the view), plus view-specific characteristics that affect the use of the entity through that view (such as mode of access to an object, formal parameter names and defaults for a subprogram, or visibility to components of a type). In most cases, a declaration also contains the definition for the entity itself (a renaming_declaration is an example of a declaration that does not define a new entity, but instead defines a view of an existing entity (see 8.5)). 7

When it is clear from context, the term *object* is used in place of *view of an object*. Similarly, the terms *type* and *subtype* are used in place of *view of a type* and *view of a subtype*, respectively. 7.1/3

8 For each declaration, the language rules define a certain region of text called the *scope* of the declaration (see 8.2). Most declarations associate an identifier with a declared entity. Within its scope, and only there, there are places where it is possible to use the identifier to refer to the declaration, the view it defines, and the associated entity; these places are defined by the visibility rules (see 8.3). At such places the identifier is said to be a *name* of the entity (the direct_name or selector_name); the name is said to *denote* the declaration, the view, and the associated entity (see 8.6). The declaration is said to *declare* the name, the view, and in most cases, the entity itself.

9 As an alternative to an identifier, an enumeration literal can be declared with a character_literal as its name (see 3.5.1), and a function can be declared with an operator_symbol as its name (see 6.1).

10 The syntax rules use the terms defining_identifier, defining_character_literal, and defining_operator_-symbol for the defining occurrence of a name; these are collectively called *defining names*. The terms direct_name and selector_name are used for usage occurrences of identifiers, character_literals, and operator_symbols. These are collectively called *usage names*.

Dynamic Semantics

11 The process by which a construct achieves its run-time effect is called *execution*. This process is also called *elaboration* for declarations and *evaluation* for expressions. One of the terms execution, elaboration, or evaluation is defined by this Reference Manual for each construct that has a run-time effect.

12 NOTE At compile time, the declaration of an entity *declares* the entity. At run time, the elaboration of the declaration *creates* the entity.

3.2 Types and Subtypes

Static Semantics

1 A *type* is characterized by a set of values, and a set of *primitive operations* which implement the fundamental aspects of its semantics. An *object* of a given type is a run-time entity that contains (has) a value of the type.

2/5 Types are grouped into *categories* of types. There exist several *language-defined categories* of types (summarized in the NOTE below), reflecting the similarity of their values and primitive operations. Most categories of types form *classes* of types. *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

3 The elementary types are the *scalar* types (*discrete* and *real*) and the *access* types (whose values provide access to objects or subprograms). Discrete types are either *integer* types or are defined by enumeration of their values (*enumeration* types). Real types are either *floating point* types or *fixed point* types.

4/2 The composite types are the *record* types, *record extensions*, *array* types, *interface* types, *task* types, and *protected* types.

4.1/2 There can be multiple views of a type with varying sets of operations. An *incomplete* type represents an incomplete view (see 3.10.1) of a type with a very restricted usage, providing support for recursive data structures. A *private* type or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. The full view (see 3.2.1) of a type represents its complete definition. An incomplete or partial view is considered a composite type, even if the full view is not.

Certain composite types (and views thereof) have special components called *discriminants* whose values affect the presence, constraints, or initialization of other components. Discriminants can be thought of as parameters of the type. 5/2

The term *subcomponent* is used in this Reference Manual in place of the term component to indicate either a component, or a component of another subcomponent. Where other subcomponents are excluded, the term component is used instead. Similarly, a *part* of an object or value is used to mean the whole object or value, or any set of its subcomponents. The terms component, subcomponent, and part are also applied to a type meaning the component, subcomponent, or part of objects and values of the type. 6/2

The set of possible values for an object of a given type can be subjected to a condition that is called a *constraint* (the case of a *null constraint* that specifies no restriction is also included); the rules for which values satisfy a given kind of constraint are given in 3.5 for range_constraints, 3.6.1 for index_constraints, and 3.7.1 for discriminant_constraints. The set of possible values for an object of an access type can also be subjected to a condition that excludes the null value (see 3.10). 7/2

A *subtype* of a given type is a combination of the type, a constraint on values of the type, and certain attributes specific to the subtype. The given type is called the *type of the subtype*. Similarly, the associated constraint is called the *constraint of the subtype*. The set of values of a subtype consists of the values of its type that satisfy its constraint and any exclusion of the null value. Such values *belong* to the subtype. The other values of the type are *outside* the subtype. 8/5

A subtype is called an *unconstrained* subtype if its type has unknown discriminants, or if its type allows range, index, or discriminant constraints, but the subtype does not impose such a constraint; otherwise, the subtype is called a *constrained* subtype (since it has no unconstrained characteristics). 9

NOTE Any set of types can be called a “category” of types, and any set of types that is closed under derivation (see 3.4) can be called a “class” of types. However, only certain categories and classes are used in the description of the rules of the language — generally those that have their own particular set of primitive operations (see 3.2.3), or that correspond to a set of types that are matched by a given kind of generic formal type (see 12.5). The following are examples of “interesting” *language-defined classes*: elementary, scalar, discrete, enumeration, character, boolean, integer, signed integer, modular, real, floating point, fixed point, ordinary fixed point, decimal fixed point, numeric, access, access-to-object, access-to-subprogram, composite, array, string, (untagged) record, tagged, task, protected, nonlimited. Special syntax is provided to define types in each of these classes. In addition to these classes, the following are examples of “interesting” *language-defined categories*: abstract, incomplete, interface, limited, private, record. 10/2

These language-defined categories are organized like this: 11/2

12/2

- all types
  - elementary
    - scalar
      - discrete
        - enumeration
          - character
          - boolean
          - other enumeration
        - integer
          - signed integer
          - modular integer
      - real
        - floating point
        - fixed point
          - ordinary fixed point
          - decimal fixed point
    - access
      - access-to-object
      - access-to-subprogram
  - composite

untagged
    array
        string
        other array
    record
    task
    protected
tagged (including interfaces)
    nonlimited tagged record
    limited tagged
        limited tagged record
        synchronized tagged
            tagged task
            tagged protected

13/2 There are other categories, such as “numeric” and “discriminated”, which represent other categorization dimensions, but do not fit into the above strictly hierarchical picture.

3.2.1 Type Declarations

1 A type_declaration declares a type and its first subtype.

Syntax

2
```
type_declaration ::= full_type_declaration
  | incomplete_type_declaration
  | private_type_declaration
  | private_extension_declaration
```

3/3
```
full_type_declaration ::=
    type defining_identifier [known_discriminant_part] is type_definition
      [aspect_specification];
  | task_type_declaration
  | protected_type_declaration
```

4/2
```
type_definition ::=
    enumeration_type_definition    | integer_type_definition
  | real_type_definition           | array_type_definition
  | record_type_definition         | access_type_definition
  | derived_type_definition        | interface_type_definition
```

Legality Rules

5 A given type shall not have a subcomponent whose type is the given type itself.

Static Semantics

6/5 The defining_identifier of a type_declaration denotes the *first subtype* of the type. The known_discriminant_part, if any, defines the discriminants of the type (see 3.7). The remainder of the type_declaration defines the remaining characteristics of (the view of) the type.

7/2 A type defined by a type_declaration is a *named* type; such a type has one or more nameable subtypes. Certain other forms of declaration also include type definitions as part of the declaration for an object. The type defined by such a declaration is *anonymous* — it has no nameable subtypes. For explanatory purposes, this document sometimes refers to an anonymous type by a pseudo-name, written in italics, and

uses such pseudo-names at places where the syntax normally requires an identifier. For a named type whose first subtype is T, this document sometimes refers to the type of T as simply "the type T".

A named type that is declared by a full_type_declaration, or an anonymous type that is defined by an access_definition or as part of declaring an object of the type, is called a *full type*. The declaration of a full type also declares the *full view* of the type. The type_definition, task_definition, protected_definition, or access_definition that defines a full type is called a *full type definition*. Types declared by other forms of type_declaration are not separate types; they are partial or incomplete views of some full type. 8/2

The definition of a type implicitly declares certain *predefined operators* that operate on the type, according to what classes the type belongs, as specified in 4.5. 9/5

The *predefined types* (for example the types Boolean, Wide_Character, Integer, *root_integer*, and *universal_integer*) are the types that are defined in a predefined library package called Standard; this package also includes the (implicit) declarations of their predefined operators. The package Standard is described in A.1. 10

Dynamic Semantics

The elaboration of a full_type_declaration consists of the elaboration of the full type definition. Each elaboration of a full type definition creates a distinct type and its first subtype. 11

Examples

Examples of type definitions: 12

```
(White, Red, Yellow, Green, Blue, Brown, Black)
range 1 .. 72
array(1 .. 10) of Integer
```
13

Examples of type declarations: 14

```
type Color  is (White, Red, Yellow, Green, Blue, Brown, Black);
type Column is range 1 .. 72;
type Table  is array(1 .. 10) of Integer;
```
15

NOTE Each of the above examples declares a named type. The identifier given denotes the first subtype of the type. Other named subtypes of the type can be declared with subtype_declarations (see 3.2.2). Although names do not directly denote types, a phrase like "the type Column" is sometimes used in this document to refer to the type of Column, where Column denotes the first subtype of the type. For an example of the definition of an anonymous type, see the declaration of the array Color_Table in 3.3.1; its type is anonymous — it has no nameable subtypes. 16

3.2.2 Subtype Declarations

A subtype_declaration declares a subtype of some previously declared type, as defined by a subtype_indication. 1

Syntax

subtype_declaration ::= 2/3
  subtype defining_identifier **is** subtype_indication
    [aspect_specification];

subtype_indication ::= [null_exclusion] subtype_mark [constraint] 3/2

subtype_mark ::= *subtype*_name 4

constraint ::= scalar_constraint | composite_constraint 5

6 scalar_constraint ::=
range_constraint | digits_constraint | delta_constraint

7 composite_constraint ::=
index_constraint | discriminant_constraint

Name Resolution Rules

8 A subtype_mark shall resolve to denote a subtype. The type *determined by* a subtype_mark is the type of the subtype denoted by the subtype_mark.

Dynamic Semantics

9 The elaboration of a subtype_declaration consists of the elaboration of the subtype_indication. The elaboration of a subtype_indication creates a new subtype. If the subtype_indication does not include a constraint, the new subtype has the same (possibly null) constraint as that denoted by the subtype_mark. The elaboration of a subtype_indication that includes a constraint proceeds as follows:

10 • The constraint is first elaborated.

11 • A check is then made that the constraint is *compatible* with the subtype denoted by the subtype_mark.

12 The condition imposed by a constraint is the condition obtained after elaboration of the constraint. The rules defining compatibility are given for each form of constraint in the appropriate subclause. These rules are such that if a constraint is *compatible* with a subtype, then the condition imposed by the constraint cannot contradict any condition already imposed by the subtype on its values. The exception Constraint_Error is raised if any check of compatibility fails.

13/5 NOTE A scalar_constraint can be applied to a subtype of an appropriate scalar type (see 3.5, 3.5.9, and J.3), even if the subtype is already constrained. On the other hand, a composite_constraint can be applied to a composite subtype (or an access-to-composite subtype) only if the composite subtype is unconstrained (see 3.6.1 and 3.7.1).

Examples

14 *Examples of subtype declarations:*

15/2
```
subtype Rainbow   is Color range Red .. Blue;      -- see 3.2.1
subtype Red_Blue  is Rainbow;
subtype Int       is Integer;
subtype Small_Int is Integer range -10 .. 10;
subtype Up_To_K   is Column range 1 .. K;          -- see 3.2.1
subtype Square    is Matrix(1 .. 10, 1 .. 10);     -- see 3.6
subtype Male      is Person(Sex => M);             -- see 3.10.1
subtype Binop_Ref is not null Binop_Ptr;           -- see 3.10
```

3.2.3 Classification of Operations

Static Semantics

1/2 An operation *operates on a type T* if it yields a value of type *T*, if it has an operand whose expected type (see 8.6) is *T*, or if it has an access parameter or access result type (see 6.1) designating *T*. A predefined operator, or other language-defined operation such as assignment or a membership test, that operates on a type, is called a *predefined operation* of the type. The *primitive operations* of a type are the predefined operations of the type, plus any user-defined primitive subprograms.

2 The *primitive subprograms* of a specific type are defined as follows:

3 • The predefined operators of the type (see 4.5);

- For a derived type, the inherited (see 3.4) user-defined subprograms; 4
- For an enumeration type, the enumeration literals (which are considered parameterless functions — see 3.5.1); 5
- For a specific type declared immediately within a package_specification, any subprograms (in addition to the enumeration literals) that are explicitly declared immediately within the same package_specification and that operate on the type; 6
- For a specific type with an explicitly declared primitive "=" operator whose result type is Boolean, the corresponding "/=" operator (see 6.6); 6.1/3
- For a nonformal type, any subprograms not covered above that are explicitly declared immediately within the same declarative region as the type and that override (see 8.3) other implicitly declared primitive subprograms of the type. 7/2

A primitive subprogram whose designator is an operator_symbol is called a *primitive operator*. A primitive function of *T* whose result type is *T* is called a *constructor function* of *T*. 8/6

3.2.4 Subtype Predicates

The language-defined *predicate aspects* Static_Predicate and Dynamic_Predicate may be used to define properties of subtypes. A *predicate specification* is an aspect_specification for one of the two predicate aspects. General rules for aspects and aspect_specifications are found in Clause 13 (13.1 and 13.1.1 respectively). The predicate aspects are assertion aspects (see 11.4.2). The predicate aspects are not inherited, but their effects are additive, as defined below. 1/5

Name Resolution Rules

The expected type for a predicate aspect expression is any boolean type. 2/3

Static Semantics

A predicate specification may be given on a type_declaration or a subtype_declaration, and applies to the declared subtype. In addition, predicate specifications apply to certain other subtypes: 3/3

- For a (first) subtype defined by a type declaration, any predicates of parent or progenitor subtypes apply. 4/4
- For a subtype created by a subtype_indication, the predicate of the subtype denoted by the subtype_mark applies. 5/3

This paragraph was deleted. 6/4

Predicate checks are defined to be *enabled* or *disabled* for a given subtype as follows: 7/3

- If a subtype is declared by a type_declaration or subtype_declaration that includes a predicate specification, then: 8/3
  - if performing checks is required by the Static_Predicate assertion policy (see 11.4.2) and the declaration includes a Static_Predicate specification, then predicate checks are enabled for the subtype; 9/3
  - if performing checks is required by the Dynamic_Predicate assertion policy (see 11.4.2) and the declaration includes a Dynamic_Predicate specification, then predicate checks are enabled for the subtype; 10/3
  - otherwise, predicate checks are disabled for the subtype, regardless of whether predicate checking is enabled for any other subtypes mentioned in the declaration; 11/3

12/4 • If a subtype is defined by a type declaration that does not include a predicate specification, then predicate checks are enabled for the subtype if and only if any predicate checks are enabled for parent or progenitor subtypes;

13/3 • If a subtype is created by a subtype_indication other than in one of the previous cases, then predicate checks are enabled for the subtype if and only if predicate checks are enabled for the subtype denoted by the subtype_mark;

14/3 • Otherwise, predicate checks are disabled for the given subtype.

14.1/4 For a subtype with a directly-specified predicate aspect, the following additional language-defined aspect may be specified with an aspect_specification (see 13.1.1):

14.2/4 Predicate_Failure

This aspect shall be specified by an expression, which determines the action to be performed when a predicate check fails because a directly-specified predicate aspect of the subtype evaluates to False, as explained below.

Name Resolution Rules

14.3/4 The expected type for the Predicate_Failure expression is String.

Legality Rules

15/3 The expression of a Static_Predicate specification shall be *predicate-static*; that is, one of the following:

16/3 • a static expression;

17/4 • a membership test whose *tested*_simple_expression is the current instance, and whose membership_choice_list meets the requirements for a static membership test (see 4.9);

18/3 • a case_expression whose *selecting*_expression is the current instance, and whose *dependent*_expressions are static expressions;

19/3 • a call to a predefined equality or ordering operator, where one operand is the current instance, and the other is a static expression;

20/4 • a call to a predefined boolean operator **and**, **or**, **xor**, or **not**, where each operand is predicate-static;

21/3 • a short-circuit control form where both operands are predicate-static; or

22/3 • a parenthesized predicate-static expression.

23/3 A predicate shall not be specified for an incomplete subtype.

24/3 If a predicate applies to a subtype, then that predicate shall not mention any other subtype to which the same predicate applies.

25/3 An index subtype, discrete_range of an index_constraint or slice, or a discrete_subtype_definition of a constrained_array_definition, entry_declaration, or entry_index_specification shall not denote a subtype to which predicate specifications apply.

26/3 The prefix of an attribute_reference whose attribute_designator is First, Last, or Range shall not denote a scalar subtype to which predicate specifications apply.

27/3 The discrete_subtype_definition of a loop_parameter_specification shall not denote a nonstatic subtype to which predicate specifications apply or any subtype to which Dynamic_Predicate specifications apply.

28/3 The discrete_choice of a named_array_aggregate shall not denote a nonstatic subtype to which predicate specifications apply.

In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 29/3

Dynamic Semantics

If any of the above Legality Rules is violated in an instance of a generic unit, Program_Error is raised at the point of the violation. 29.1/4

To determine whether a value *satisfies the predicates* of a subtype *S*, the following tests are performed in the following order, until one of the tests fails, in which case the predicates are not satisfied and no further tests are performed, or all of the tests succeed, in which case the predicates are satisfied: 29.2/4

- the value is first tested to determine whether it satisfies any constraints or any null exclusion of *S*; 29.3/4
- then: 29.4/4
  - if *S* is a first subtype, the value is tested to determine whether it satisfies the predicates of the parent and progenitor subtypes (if any) of *S* (in an arbitrary order), after a (view) conversion of the value to the corresponding parent or progenitor type; 29.5/5
  - if *S* is defined by a subtype_indication, the value is tested to determine whether it satisfies the predicates of the subtype denoted by the subtype_mark of the subtype_indication; 29.6/4
- finally, if *S* is defined by a declaration to which one or more predicate specifications apply, the predicates are evaluated (in an arbitrary order) to test that all of them yield True for the given value. 29.7/4

If predicate checks are enabled for a given subtype, then: 30/3

On a subtype conversion, a check is performed that the operand satisfies the predicates of the target subtype, except for certain view conversions (see 4.6). In addition, after normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by reference, a check is performed that the value of the parameter satisfies the predicates of the subtype of the actual. For an object created by an object_declaration with no explicit initialization expression, or by an uninitialized allocator, if the types of any parts have specified Default_Value or Default_Component_Value aspects, or any subcomponents have default_expressions, a check is performed that the value of the created object satisfies the predicates of the nominal subtype. 31/5

If any of the predicate checks fail, Assertion_Error is raised, unless the subtype whose directly-specified predicate aspect evaluated to False also has a directly-specified Predicate_Failure aspect. In that case, the specified Predicate_Failure expression is evaluated; if the evaluation of the Predicate_Failure expression propagates an exception occurrence, then this occurrence is propagated for the failure of the predicate check; otherwise, Assertion_Error is raised, with an associated message string defined by the value of the Predicate_Failure expression. In the absence of such a Predicate_Failure aspect, an implementation-defined message string is associated with the Assertion_Error exception. 31.1/4

Paragraphs 32 and 33 were moved above

NOTE 1 A predicate specification does not cause a subtype to be considered constrained. 34/3

NOTE 2 A Static_Predicate, like a constraint, always remains True for all objects of the subtype, except in the case of uninitialized variables and other invalid values. A Dynamic_Predicate, on the other hand, is checked as specified above, but can become False at other times. For example, the predicate of a record subtype is not checked when a subcomponent is modified. 35/3

36/4 NOTE 3 No predicates apply to the base subtype of a scalar type; every value of a scalar type *T* is considered to satisfy the predicates of *T*'Base.

37/4 NOTE 4 Predicate_Failure expressions are never evaluated during the evaluation of a membership test (see 4.5.2) or Valid attribute (see 13.9.2).

38/4 NOTE 5 A Predicate_Failure expression can be a raise_expression (see 11.3).

Examples

39/5 *Examples of predicates applied to scalar types:*

40/4
```
subtype Basic_Letter is Character -- See A.3.2 for "basic letter".
   with Static_Predicate => Basic_Letter in 'A'..'Z' | 'a'..'z' | 'Æ' |
                            'æ' | 'Ð' | 'ð' | 'Þ' | 'þ' | 'ß';
```

41/4
```
subtype Even_Integer is Integer
   with Dynamic_Predicate => Even_Integer mod 2 = 0,
        Predicate_Failure => "Even_Integer must be a multiple of 2";
```

42/4 *Text_IO (see A.10.1) could have used predicates to describe some common exceptional conditions as follows:*

43/4
```
with Ada.IO_Exceptions;
package Ada.Text_IO is
```

44/4
```
   type File_Type is limited private;
```

45/4
```
   subtype Open_File_Type is File_Type
      with Dynamic_Predicate => Is_Open (Open_File_Type),
           Predicate_Failure => raise Status_Error with "File not open";
   subtype Input_File_Type is Open_File_Type
      with Dynamic_Predicate => Mode (Input_File_Type) = In_File,
           Predicate_Failure => raise Mode_Error
              with "Cannot read file: " & Name (Input_File_Type);
   subtype Output_File_Type is Open_File_Type
      with Dynamic_Predicate => Mode (Output_File_Type) /= In_File,
           Predicate_Failure => raise Mode_Error
              with "Cannot write file: " & Name (Output_File_Type);
```

46/4
```
   ...
```

47/4
```
   function Mode (File : in Open_File_Type) return File_Mode;
   function Name (File : in Open_File_Type) return String;
   function Form (File : in Open_File_Type) return String;
```

48/4
```
   ...
```

49/4
```
   procedure Get (File : in Input_File_Type; Item : out Character);
```

50/4
```
   procedure Put (File : in Output_File_Type; Item : in Character);
```

51/4
```
   ...
```

52/4
```
   -- Similarly for all of the other input and output subprograms.
```

3.3 Objects and Named Numbers

1 Objects are created at run time and contain a value of a given type. An object can be created and initialized as part of elaborating a declaration, evaluating an allocator, aggregate, or function_call, or passing a parameter by copy. Prior to reclaiming the storage for an object, it is finalized if necessary (see 7.6.1).

Static Semantics

2 All of the following are objects:

3 • the entity declared by an object_declaration;

4 • a formal parameter of a subprogram, entry, or generic subprogram;

5 • a generic formal object;

- a loop parameter; 6
- the index parameter of an iterated_component_association; 6.1/5
- the chunk parameter of a chunk_specification; 6.2/5
- a choice parameter of an exception_handler; 7
- an entry index of an entry_body; 8
- the result of dereferencing an access-to-object value (see 4.1); 9
- the return object of a function; 10/3
- the result of evaluating an aggregate; 11
- a value conversion or qualified_expression whose operand denotes an object; 11.1/5
- a component, slice, or view conversion of another object. 12

An object is either a *constant* object or a *variable* object. Similarly, a view of an object is either a *constant* or a *variable*. All views of a constant elementary object are constant. All views of a constant composite object are constant, except for parts that are of controlled or immutably limited types; variable views of those parts and their subcomponents may exist. In this sense, objects of controlled and immutably limited types are *inherently mutable*. A constant view of an object cannot be used to modify its value. The terms constant and variable by themselves refer to constant and variable views of objects. 13/3

A constant object is *known to have no variable views* if it does not have a part that is immutably limited, or of a controlled type, private type, or private extension. 13.1/5

The value of an object is *read* when the value of any part of the object is evaluated, or when the value of an enclosing object is evaluated. The value of a variable is *updated* when an assignment is performed to any part of the variable, or when an assignment is performed to an enclosing object. 14

Whether a view of an object is constant or variable is determined by the definition of the view. The following (and no others) represent variables: 15/5

- an object declared by an object_declaration without the reserved word **constant**; 16/5
- a formal parameter of mode **in out** or **out**; 17/5
- a generic formal object of mode **in out**; 17.1/5
- a non-discriminant component of a variable; 18/5
- a slice of a variable; 18.1/5
- a loop parameter that is specified to be a variable for a generalized loop (see 5.5.2); 18.2/5
- a view conversion of a variable; 19/5
- a dereference of an access-to-variable value; 20/5
- the return object declared by an extended_return_statement without the reserved word **constant**; 20.1/5
- the current instance of a type other than a protected type, if the current instance is an object and not a value (see 8.6); 21/5
- *This paragraph was deleted.* 21.1/5
- the current instance of a protected unit except within the body of a protected function of that protected unit, or within a function declared immediately within the body of the protected unit; 21.2/5

22/5 • an attribute_reference where the attribute is defined to denote a variable (for example, the Storage_Pool attribute – see 13.11).

23/5 At the place where a view of an object is defined, a *nominal subtype* is associated with the view. The *nominal type* of a view is the type of the nominal subtype of the view. The object's *actual subtype* (that is, its subtype) can be more restrictive than the nominal subtype of the view; it always is more restrictive if the nominal subtype is an *indefinite subtype*. A subtype is an indefinite subtype if it is an unconstrained array subtype, or if it has unknown discriminants or unconstrained discriminants without defaults (see 3.7); otherwise, the subtype is a *definite* subtype (all elementary subtypes are definite subtypes). A class-wide subtype is defined to have unknown discriminants, and is therefore an indefinite subtype. An indefinite subtype does not by itself provide enough information to create an object; an additional constraint or explicit initialization expression is necessary (see 3.3.1). A component cannot have an indefinite nominal subtype.

23.1/3 A view of a composite object is *known to be constrained* if:

23.2/5 • its nominal subtype is constrained and not an untagged partial view, and it is neither a value conversion nor a qualified_expression; or

23.3/3 • its nominal subtype is indefinite; or

23.4/3 • its type is immutably limited (see 7.5); or

23.5/3 • it is part of a stand-alone constant (including a generic formal object of mode **in**); or

23.6/3 • it is part of a formal parameter of mode **in**; or

23.7/3 • it is part of the object denoted by a function_call or aggregate; or

23.8/5 • it is a value conversion or qualified_expression where the operand denotes a view of a composite object that is known to be constrained; or

23.9/5 • it is part of a constant return object of an extended_return_statement; or

23.10/5 • it is a dereference of a pool-specific access type, and there is no ancestor of its type that has a constrained partial view.

23.11/5 For the purposes of determining within a generic body whether an object is known to be constrained:

23.12/5 • if a subtype is a descendant of an untagged generic formal private or derived type, and the subtype is not an unconstrained array subtype, it is not considered indefinite and is considered to have a constrained partial view;

23.13/5 • if a subtype is a descendant of a formal access type, it is not considered pool-specific.

24 A *named number* provides a name for a numeric value known at compile time. It is declared by a number_declaration.

25 NOTE 1 A constant cannot be the target of an assignment operation, nor be passed as an **in out** or **out** parameter, between its initialization and finalization, if any.

25.1/3 NOTE 2 The value of a constant object cannot be changed after its initialization, except in some cases where the object has a controlled or immutably limited part (see 7.5, 7.6, and 13.9.1).

26/3 NOTE 3 The nominal and actual subtypes of an elementary object are always the same. For a discriminated or array object, if the nominal subtype is constrained, then so is the actual subtype.

3.3.1 Object Declarations

An object_declaration declares a *stand-alone* object with a given nominal subtype and, optionally, an explicit initial value given by an initialization expression. For an array, access, task, or protected object, the object_declaration may include the definition of the (anonymous) type of the object. 1/3

Syntax

```
object_declaration ::=                                                          2/3
   defining_identifier_list : [aliased] [constant] subtype_indication [:= expression]
      [aspect_specification];
 | defining_identifier_list : [aliased] [constant] access_definition [:= expression]
      [aspect_specification];
 | defining_identifier_list : [aliased] [constant] array_type_definition [:= expression]
      [aspect_specification];
 | single_task_declaration
 | single_protected_declaration

defining_identifier_list ::=                                                    3
  defining_identifier {, defining_identifier}
```

Name Resolution Rules

For an object_declaration with an expression following the compound delimiter :=, the type expected for the expression is that of the object. This expression is called the *initialization expression*. 4

Legality Rules

An object_declaration without the reserved word **constant** declares a variable object. If it has a subtype_indication or an array_type_definition that defines an indefinite subtype, then there shall be an initialization expression. 5/2

Static Semantics

An object_declaration with the reserved word **constant** declares a constant object. If it has an initialization expression, then it is called a *full constant declaration*. Otherwise, it is called a *deferred constant declaration*. The rules for deferred constant declarations are given in 7.4. The rules for full constant declarations are given in this subclause. 6/5

Any declaration that includes a defining_identifier_list with more than one defining_identifier is equivalent to a series of declarations each containing one defining_identifier from the list, with the rest of the text of the declaration copied for each declaration in the series, in the same order as the list. The remainder of this Reference Manual relies on this equivalence; explanations are given for declarations with a single defining_identifier. 7

The subtype_indication, access_definition, or full type definition of an object_declaration defines the nominal subtype of the object. The object_declaration declares an object of the type of the nominal subtype. 8/2

A component of an object is said to *require late initialization* if: 8.1/5

- it has an access discriminant value constrained by a per-object expression; or 8.2/5
- it has an initialization expression that includes a name denoting an access discriminant; or 8.3/5

8.4/5 • it has an initialization expression that includes a reference to the current instance of the type either by name or implicitly as the target object of a call.

Dynamic Semantics

9/2 If a composite object declared by an object_declaration has an unconstrained nominal subtype, then if this subtype is indefinite or the object is constant the actual subtype of this object is constrained. The constraint is determined by the bounds or discriminants (if any) of its initial value; the object is said to be *constrained by its initial value*. When not constrained by its initial value, the actual and nominal subtypes of the object are the same. If its actual subtype is constrained, the object is called a *constrained object*.

10 For an object_declaration without an initialization expression, any initial values for the object or its subcomponents are determined by the *implicit initial values* defined for its nominal subtype, as follows:

11 • The implicit initial value for an access subtype is the null value of the access type.

11.1/5 • The implicit initial value for a scalar subtype that has the Default_Value aspect specified is the value of that aspect converted to the nominal subtype (which can raise Constraint_Error — see 4.6, “Type Conversions”);

12 • The implicit initial (and only) value for each discriminant of a constrained discriminated subtype is defined by the subtype.

13/5 • For a (definite) composite subtype, the implicit initial value of each component with a default_expression is obtained by evaluation of this expression and conversion to the component's nominal subtype (which can raise Constraint_Error), unless the component is a discriminant of a constrained subtype (the previous case), or is in an excluded variant (see 3.8.1). For each component that does not have a default_expression, if the composite subtype has the Default_Component_Value aspect specified, the implicit initial value is the value of that aspect converted to the component's nominal subtype; otherwise, any implicit initial values are those determined by the component's nominal subtype.

14 • For a protected or task subtype, there is an implicit component (an entry queue) corresponding to each entry, with its implicit initial value being an empty queue.

15 The elaboration of an object_declaration proceeds in the following sequence of steps:

16/2 1. The subtype_indication, access_definition, array_type_definition, single_task_declaration, or single_protected_declaration is first elaborated. This creates the nominal subtype (and the anonymous type in the last four cases).

17/5 2. If the object_declaration includes an initialization expression, the (explicit) initial value is obtained by evaluating the expression and converting it to the nominal subtype (which can raise Constraint_Error — see 4.6).

18/2 3. The object is created, and, if there is not an initialization expression, the object is *initialized by default*. When an object is initialized by default, any per-object constraints (see 3.8) are elaborated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype. Any initial values (whether explicit or implicit) are assigned to the object or to the corresponding subcomponents. As described in 5.2 and 7.6, Initialize and Adjust procedures can be called.

19/2 *This paragraph was deleted.*

20/2 For the third step above, evaluations and assignments are performed in an arbitrary order subject to the following restrictions:

20.1/2 • Assignment to any part of the object is preceded by the evaluation of the value that is to be assigned.

- The evaluation of a default_expression that includes the name of a discriminant is preceded by the assignment to that discriminant. 20.2/2
- The evaluation of the default_expression for any component that depends on a discriminant is preceded by the assignment to that discriminant. 20.3/2
- The assignments to any components, including implicit components, not requiring late initialization precede the initial value evaluations for any components requiring late initialization; if two components both require late initialization, then assignments to parts of the component occurring earlier in the order of the component declarations precede the initial value evaluations of the component occurring later. 20.4/3

There is no implicit initial value defined for a scalar subtype unless the Default_Value aspect has been specified for the type. In the absence of an explicit initialization or the specification of the Default_Value aspect, a newly created scalar object can have a value that does not belong to its subtype (see 13.9.1 and H.1). 21/5

NOTE 1 Implicit initial values are not defined for an indefinite subtype, because if an object's nominal subtype is indefinite, an explicit initial value is required. 22

NOTE 2 As indicated above, a stand-alone object is an object declared by an object_declaration. Similar definitions apply to "stand-alone constant" and "stand-alone variable". A subcomponent of an object is not a stand-alone object, nor is an object that is created by an allocator. An object declared by a loop_parameter_specification, iterator_specification, iterated_component_association, chunk_specification, parameter_specification, entry_index_specification, choice_parameter_specification, extended_return_statement, or a formal_object_declaration of mode **in out** is not considered a stand-alone object. 23/5

NOTE 3 The type of a stand-alone object cannot be abstract (see 3.9.3). 24

Examples

Example of a multiple object declaration: 25

```
-- the multiple object declaration                                          26
John, Paul : not null Person_Name := new Person(Sex => M); -- see 3.10.1    27/2
-- is equivalent to the two single object declarations in the order given   28
John : not null Person_Name := new Person(Sex => M);                        29/2
Paul : not null Person_Name := new Person(Sex => M);
```

Examples of variable declarations: 30

```
Count, Sum  : Integer;                                                      31/5
Size        : Integer range 0 .. 10_000 := 0;
Sorted      : Boolean := False;
Color_Table : array(1 .. Max) of Color;
Option      : Bit_Vector(1 .. 10) := (others => True); -- see 3.6
Hello       : aliased String := "Hi, world.";
θ, φ        : Float range -π .. +π;
```

Examples of constant declarations: 32

```
Limit     : constant Integer := 10_000;                                     33/5
Low_Limit : constant Integer := Limit/10;
Tolerance : constant Real := Dispersion(1.15);
A_String  : constant String := "A";
Hello_Msg : constant access String := Hello'Access; -- see 3.10.2
```

3.3.2 Number Declarations

1 A number_declaration declares a named number.

Syntax

2 number_declaration ::=
defining_identifier_list : **constant** := *static_*expression;

Name Resolution Rules

3 The *static_*expression given for a number_declaration is expected to be of any numeric type.

3.1/5 A name that denotes a number_declaration is interpreted as a value of a universal type, unless the expected type for the name is a non-numeric type with an Integer_Literal or Real_Literal aspect, in which case it is interpreted to be of its expected type.

Legality Rules

4/5 The *static_*expression given for a number declaration shall be a static expression, as defined by 4.9.

Static Semantics

5 The named number denotes a value of type *universal_integer* if the type of the *static_*expression is an integer type. The named number denotes a value of type *universal_real* if the type of the *static_*expression is a real type.

6 The value denoted by the named number is the value of the *static_*expression, converted to the corresponding universal type.

Dynamic Semantics

7 The elaboration of a number_declaration has no effect.

Examples

8 *Examples of number declarations:*

9
```
Two_Pi        : constant := 2.0*Ada.Numerics.Pi;
                                          -- a real number (see A.5)
```

10/2
```
Max           : constant := 500;          -- an integer number
Max_Line_Size : constant := Max/6;        -- the integer 83
Power_16      : constant := 2**16;        -- the integer 65_536
One, Un, Eins : constant := 1;            -- three different names for 1
```

3.4 Derived Types and Classes

1/2 A derived_type_definition defines a *derived type* (and its first subtype) whose characteristics are *derived* from those of a parent type, and possibly from progenitor types.

1.1/2 A *class of types* is a set of types that is closed under derivation; that is, if the parent or a progenitor type of a derived type belongs to a class, then so does the derived type. By saying that a particular group of types forms a class, we are saying that all derivatives of a type in the set inherit the characteristics that define that set. The more general term *category of types* is used for a set of types whose defining characteristics are not necessarily inherited by derivatives; for example, limited, abstract, and interface are all categories of types, but not classes of types.

Syntax

derived_type_definition ::= 2/2
[**abstract**] [**limited**] **new** *parent*_subtype_indication [[**and** interface_list] record_extension_part]

Legality Rules

The *parent*_subtype_indication defines the *parent subtype*; its type is the *parent type*. The interface_list defines the progenitor types (see 3.9.4). A derived type has one parent type and zero or more progenitor types. 3/2

A type shall be completely defined (see 3.11.1) prior to being specified as the parent type in a derived_type_definition — the full_type_declarations for the parent type and any of its subcomponents have to precede the derived_type_definition. 4

If there is a record_extension_part, the derived type is called a *record extension* of the parent type. A record_extension_part shall be provided if and only if the parent type is a tagged type. An interface_list shall be provided only if the parent type is a tagged type. 5/2

If the reserved word **limited** appears in a derived_type_definition, the parent type shall be a limited type. If the parent type is a tagged formal type, then in addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 5.1/3

Static Semantics

The first subtype of the derived type is unconstrained if a known_discriminant_part is provided in the declaration of the derived type, or if the parent subtype is unconstrained. Otherwise, the constraint of the first subtype *corresponds* to that of the parent subtype in the following sense: it is the same as that of the parent subtype except that for a range constraint (implicit or explicit), the value of each bound of its range is replaced by the corresponding value of the derived type. 6

The first subtype of the derived type excludes null (see 3.10) if and only if the parent subtype excludes null. 6.1/2

The *characteristics* and implicitly declared primitive subprograms of the derived type are defined as follows: 7/3

- If the parent type or a progenitor type belongs to a class of types, then the derived type also belongs to that class. The following sets of types, as well as any higher-level sets composed from them, are classes in this sense, and hence the characteristics defining these classes are inherited by derived types from their parent or progenitor types: signed integer, modular integer, ordinary fixed, decimal fixed, floating point, enumeration, boolean, character, access-to-constant, general access-to-variable, pool-specific access-to-variable, access-to-subprogram, array, string, non-array composite, nonlimited, untagged record, tagged, task, protected, and synchronized tagged. 8/2
- If the parent type is an elementary type or an array type, then the set of possible values of the derived type is a copy of the set of possible values of the parent type. For a scalar type, the base range of the derived type is the same as that of the parent type. 9
- If the parent type is a composite type other than an array type, then the components, protected subprograms, and entries that are declared for the derived type are as follows: 10
  - The discriminants specified by a new known_discriminant_part, if there is one; otherwise, each discriminant of the parent type (implicitly declared in the same order with the same specifications) — in the latter case, the discriminants are said to be *inherited*, or if unknown in the parent, are also unknown in the derived type; 11

12 • Each nondiscriminant component, entry, and protected subprogram of the parent type, implicitly declared in the same order with the same declarations; these components, entries, and protected subprograms are said to be *inherited*;

13 • Each component declared in a record_extension_part, if any.

14 Declarations of components, protected subprograms, and entries, whether implicit or explicit, occur immediately within the declarative region of the type, in the order indicated above, following the parent subtype_indication.

15/2 • *This paragraph was deleted.*

16 • For each predefined operator of the parent type, there is a corresponding predefined operator of the derived type.

17/2 • For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent type or of a progenitor type that already exists at the place of the derived_type_definition, there exists a corresponding *inherited* primitive subprogram of the derived type with the same defining name. Primitive user-defined equality operators of the parent type and any progenitor types are also inherited by the derived type, except when the derived type is a nonlimited record extension, and the inherited operator would have a profile that is type conformant with the profile of the corresponding predefined equality operator; in this case, the user-defined equality operator is not inherited, but is rather incorporated into the implementation of the predefined equality operator of the record extension (see 4.5.2).

18/3 The profile of an inherited subprogram (including an inherited enumeration literal) is obtained from the profile of the corresponding (user-defined) primitive subprogram of the parent or progenitor type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent or progenitor type, other than those subtypes found in the designated profile of an access_definition, with a *corresponding subtype* of the derived type. For a given subtype of the parent or progenitor type, the corresponding subtype of the derived type is defined as follows:

19 • If the declaration of the derived type has neither a known_discriminant_part nor a record_extension_part, then the corresponding subtype has a constraint that corresponds (as defined above for the first subtype of the derived type) to that of the given subtype.

20 • If the derived type is a record extension, then the corresponding subtype is the first subtype of the derived type.

21 • If the derived type has a new known_discriminant_part but is not a record extension, then the corresponding subtype is constrained to those values that when converted to the parent type belong to the given subtype (see 4.6).

22/2 The same formal parameters have default_expressions in the profile of the inherited subprogram. Any type mismatch due to the systematic replacement of the parent or progenitor type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.

23/2 If a primitive subprogram of the parent or progenitor type is visible at the place of the derived_type_definition, then the corresponding inherited subprogram is implicitly declared immediately after the derived_type_definition. Otherwise, the inherited subprogram is implicitly declared later or not at all, as explained in 7.3.1.

24 A derived type can also be defined by a private_extension_declaration (see 7.3) or a formal_derived_type_definition (see 12.5.1). Such a derived type is a partial view of the corresponding full or actual type.

25 All numeric types are derived types, in that they are implicitly derived from a corresponding root numeric type (see 3.5.4 and 3.5.6).

Dynamic Semantics

The elaboration of a derived_type_definition creates the derived type and its first subtype, and consists of the elaboration of the subtype_indication and the record_extension_part, if any. If the subtype_indication depends on a discriminant, then only those expressions that do not depend on a discriminant are evaluated. 26

For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent or progenitor type is performed; the normal conversion of each actual parameter to the subtype of the corresponding formal parameter (see 6.4.1) performs any necessary type conversion as well. If the result type of the inherited subprogram is the derived type, the result of calling the subprogram of the parent or progenitor is converted to the derived type, or in the case of a type that has automatic constructor extension (see 3.9.1), extended to the derived type using the equivalent of an extension_aggregate with the original result as the ancestor_part and **others** => <> as the record_component_association_list. In either case, the result is then converted to the corresponding subtype (as defined above) of the derived type. 27/6

NOTE 1 Classes are closed under derivation — any class that contains a type also contains its derivatives. Operations available for a given class of types are available for the derived types in that class. 28

NOTE 2 Evaluating an inherited enumeration literal is equivalent to evaluating the corresponding enumeration literal of the parent type, and then converting the result to the derived type. This follows from their equivalence to parameterless functions. 29

NOTE 3 A generic subprogram is not a subprogram, and hence cannot be a primitive subprogram and cannot be inherited by a derived type. On the other hand, an instance of a generic subprogram can be a primitive subprogram, and hence can be inherited. 30

NOTE 4 If the parent type is an access type, then the parent and the derived type share the same storage pool; there is a **null** access value for the derived type and it is the implicit initial value for the type. See 3.10. 31

NOTE 5 If the parent type is a boolean type, the predefined relational operators of the derived type deliver a result of the predefined type Boolean (see 4.5.2). If the parent type is an integer type, the right operand of the predefined exponentiation operator is of the predefined type Integer (see 4.5.6). 32

NOTE 6 Any discriminants of the parent type are either all inherited, or completely replaced with a new set of discriminants. 33

NOTE 7 For an inherited subprogram, the subtype of a formal parameter of the derived type can be such that it has no value in common with the first subtype of the derived type. 34/5

NOTE 8 If the reserved word **abstract** is given in the declaration of a type, the type is abstract (see 3.9.3). 35

NOTE 9 An interface type that has a progenitor type "is derived from" that type. A derived_type_definition, however, never defines an interface type. 35.1/2

NOTE 10 It is illegal for the parent type of a derived_type_definition to be a synchronized tagged type. 35.2/2

Examples

Examples of derived type declarations: 36

```
type Local_Coordinate is new Coordinate;   -- two different types
type Midweek is new Day range Tue .. Thu; -- see 3.5.1
type Counter is new Positive;              -- same range as Positive
```
37

```
type Special_Key is new Key_Manager.Key;   -- see 7.3.1
   -- the inherited subprograms have the following specifications:
   --       procedure Get_Key(K : out Special_Key);
   --       function "<"(X,Y : Special_Key) return Boolean;
```
38

3.4.1 Derivation Classes

1 In addition to the various language-defined classes of types, types can be grouped into *derivation classes*.

Static Semantics

2/2 A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. A derived type, interface type, type extension, task type, protected type, or formal derived type is also derived from every ancestor of each of its progenitor types, if any. The derivation class of types for a type *T* (also called the class *rooted* at *T*) is the set consisting of *T* (the *root type* of the class) and all types derived from *T* (directly or indirectly) plus any associated universal or class-wide types (defined below).

3/5 Every type is one of a *specific* type, a *class-wide* type, or a *universal* type. A specific type is one defined by a type_declaration, a formal_type_declaration, or a full type definition embedded in another construct. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

4 Class-wide types

Class-wide types are defined for (and belong to) each derivation class rooted at a tagged type (see 3.9). Given a subtype S of a tagged type *T*, S'Class is the subtype_mark for a corresponding subtype of the tagged class-wide type *T*'Class. Such types are called "class-wide" because when a formal parameter is defined to be of a class-wide type *T*'Class, an actual parameter of any type in the derivation class rooted at *T* is acceptable (see 8.6).

5 The set of values for a class-wide type *T*'Class is the discriminated union of the set of values of each specific type in the derivation class rooted at *T* (the tag acts as the implicit discriminant — see 3.9). Class-wide types have no primitive subprograms of their own. However, as explained in 3.9.2, operands of a class-wide type *T*'Class can be used as part of a dispatching call on a primitive subprogram of the type *T*. The only components (including discriminants) of *T*'Class that are visible are those of *T*. If S is a first subtype, then S'Class is a first subtype.

5.1/6 When a language rule references a given primitive operation of a type, in the case of a class-wide type *T*'Class, this is equivalent to a subprogram (with an intrinsic calling convention — see 6.3.1) whose body consists of a dispatching call upon the corresponding operation of *T*, with its formal parameters as the actual parameters. If it is a function, the result of the dispatching call is returned.

6/5 Universal types

Universal types are defined for (and belong to) the integer, real, fixed point, and access classes, and are referred to in this document as respectively, *universal_integer*, *universal_real*, *universal_fixed*, and *universal_access*. These are analogous to class-wide types for these language-defined elementary classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the corresponding class is acceptable. In addition, a value of a universal type (including an integer or real numeric_literal, or the literal **null**) is "universal" in that it is acceptable where some particular type in the class is expected (see 8.6).

7 The set of values of a universal type is the undiscriminated union of the set of values possible for any definable type in the associated class. Like class-wide types, universal types have no primitive subprograms of their own. However, their "universality" allows them to be used as operands with the primitive subprograms of any type in the corresponding class.

The integer and real numeric classes each have a specific root type in addition to their universal type, named respectively *root_integer* and *root_real*. 8

A class-wide or universal type is said to *cover* all of the types in its class. In addition, *universal_integer* covers a type that has a specified Integer_Literal aspect, while *universal_real* covers a type that has a specified Real_Literal aspect (see 4.2.1). A specific type covers only itself. 9/5

A specific type *T2* is defined to be a *descendant* of a type *T1* if *T2* is the same as *T1*, or if *T2* is derived (directly or indirectly) from *T1*. A class-wide type *T2*'Class is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the numeric universal types are defined to be descendants of the root types of their classes. If a type *T2* is a descendant of a type *T1*, then *T1* is called an *ancestor* of *T2*. An *ultimate ancestor* of a type is an ancestor of that type that is not itself a descendant of any other type. Every untagged type has a unique ultimate ancestor. 10/2

An inherited component (including an inherited discriminant) of a derived type is inherited *from* a given ancestor of the type if the corresponding component was inherited by each derived type in the chain of derivations going back to the given ancestor. 11

NOTE Because operands of a universal type are acceptable to the predefined operators of any type in their class, ambiguity can result. For *universal_integer* and *universal_real*, this potential ambiguity is resolved by giving a preference (see 8.6) to the predefined operators of the corresponding root types (*root_integer* and *root_real*, respectively). Hence, in an apparently ambiguous expression like 12

1 + 4 < 7 13

where each of the literals is of type *universal_integer*, the predefined operators of *root_integer* will be preferred over those of other specific integer types, thereby resolving the ambiguity. 14

3.5 Scalar Types

Scalar types comprise enumeration types, integer types, and real types. Enumeration types and integer types are called *discrete* types; each value of a discrete type has a *position number* which is an integer value. Integer types and real types are called *numeric* types. All scalar types are ordered, that is, all relational operators are predefined for their values. 1

Syntax

range_constraint ::= **range** range 2

range ::= range_attribute_reference 3
  | simple_expression .. simple_expression

A *range* has a *lower bound* and an *upper bound* and specifies a subset of the values of some scalar type (the *type of the range*). A range with lower bound L and upper bound R is described by “L .. R”. If R is less than L, then the range is a *null range*, and specifies an empty set of values. Otherwise, the range specifies the values of the type from the lower bound to the upper bound, inclusive. A value *belongs* to a range if it is of the type of the range, and is in the subset of values specified by the range. A value *satisfies* a range constraint if it belongs to the associated range. One range is *included* in another if all values that belong to the first range also belong to the second. 4

Name Resolution Rules

For a subtype_indication containing a range_constraint, either directly or as part of some other scalar_constraint, the type of the range shall resolve to that of the type determined by the subtype_mark of the subtype_indication. For a range of a given type, the simple_expressions of the range (likewise, 5

the simple_expressions of the equivalent range for a range_attribute_reference) are expected to be of the type of the range.

Static Semantics

6 The *base range* of a scalar type is the range of finite values of the type that can be represented in every unconstrained object of the type; it is also the range supported at a minimum for intermediate values during the evaluation of expressions involving predefined operators of the type.

7 A constrained scalar subtype is one to which a range constraint applies. The *range* of a constrained scalar subtype is the range associated with the range constraint of the subtype. The *range* of an unconstrained scalar subtype is the base range of its type.

Dynamic Semantics

8 A range is *compatible* with a scalar subtype if and only if it is either a null range or each bound of the range belongs to the range of the subtype. A range_constraint is *compatible* with a scalar subtype if and only if its range is compatible with the subtype.

9 The elaboration of a range_constraint consists of the evaluation of the range. The evaluation of a range determines a lower bound and an upper bound. If simple_expressions are given to specify bounds, the evaluation of the range evaluates these simple_expressions in an arbitrary order, and converts them to the type of the range. If a range_attribute_reference is given, the evaluation of the range consists of the evaluation of the range_attribute_reference.

10 *Attributes*

11 For every scalar subtype S, the following attributes are defined:

12 S'First — S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S.

13 S'Last — S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S.

14 S'Range — S'Range is equivalent to the range S'First .. S'Last.

15 S'Base — S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type.

16 S'Min — S'Min denotes a function with the following specification:

17
```
function S'Min(Left, Right : S'Base)
  return S'Base
```

18 The function returns the lesser of the values of the two parameters.

19 S'Max — S'Max denotes a function with the following specification:

20
```
function S'Max(Left, Right : S'Base)
  return S'Base
```

21 The function returns the greater of the values of the two parameters.

22 S'Succ — S'Succ denotes a function with the following specification:

23
```
function S'Succ(Arg : S'Base)
  return S'Base
```

24 For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of *Arg*. For a fixed point type, the function returns the result of adding *small* to the value of *Arg*. For a

floating point type, the function returns the machine number (as defined in 3.5.7) immediately above the value of *Arg*; Constraint_Error is raised if there is no such machine number.

S'Pred

S'Pred denotes a function with the following specification: 25

```
function S'Pred(Arg : S'Base)                                    26
  return S'Base
```

For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of *Arg*. For a fixed point type, the function returns the result of subtracting *small* from the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of *Arg*; Constraint_Error is raised if there is no such machine number. 27

Paragraphs 28 through 37 were moved to 4.10, "Image Attributes".

S'Wide_Wide_Width 37.1/5

S'Wide_Wide_Width denotes the maximum length of a Wide_Wide_String returned by S'Wide_Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

S'Wide_Width 38/5

S'Wide_Width denotes the maximum length of a Wide_String returned by S'Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*.

S'Width

S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. 39/5

S'Wide_Wide_Value 39.1/2

S'Wide_Wide_Value denotes a function with the following specification:

```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)               39.2/2
  return S'Base
```

This function returns a value given an image of the value as a Wide_Wide_String, ignoring any leading or trailing spaces. 39.3/2

For the evaluation of a call on S'Wide_Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Wide_Image for a nongraphic character of the type), the result is the corresponding enumeration value; otherwise, Constraint_Error is raised. 39.4/3

For the evaluation of a call on S'Wide_Wide_Value for an integer subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an integer literal, with an optional leading sign character (plus or minus for a signed type; only plus for a modular type), and the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise, Constraint_Error is raised. 39.5/3

For the evaluation of a call on S'Wide_Wide_Value for a real subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of one of the following: 39.6/2

- numeric_literal 39.7/2

39.8/2 - numeral.[exponent]

39.9/2 - .numeral[exponent]

39.10/2 - base#based_numeral.#[exponent]

39.11/2 - base#.based_numeral#[exponent]

39.12/3 with an optional leading sign character (plus or minus), and if the corresponding numeric value belongs to the base range of the type of S, then that value is the result; otherwise, Constraint_Error is raised. The sign of a zero value is preserved (positive if none has been specified) if S'Signed_Zeros is True.

40 S'Wide_Value

S'Wide_Value denotes a function with the following specification:

41
```
function S'Wide_Value(Arg : Wide_String)
  return S'Base
```

42 This function returns a value given an image of the value as a Wide_String, ignoring any leading or trailing spaces.

43/5 For the evaluation of a call on S'Wide_Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Wide_Image for a value of the type, assuming a default implementation of S'Put_Image), the result is the corresponding enumeration value; otherwise, Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Wide_Value with *Arg* of type Wide_String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String.

Paragraphs 44 through 51 were moved to Wide_Wide_Value.

52 S'Value S'Value denotes a function with the following specification:

53
```
function S'Value(Arg : String)
  return S'Base
```

54 This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces.

55/5 For the evaluation of a call on S'Value for an enumeration subtype S, if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it corresponds to a literal of the type of S (or corresponds to the result of S'Image for a value of the type, assuming a default implementation of S'Put_Image), the result is the corresponding enumeration value; otherwise, Constraint_Error is raised. For a numeric subtype S, the evaluation of a call on S'Value with *Arg* of type String is equivalent to a call on S'Wide_Wide_Value for a corresponding *Arg* of type Wide_Wide_String.

Implementation Permissions

56/2 An implementation may extend the Wide_Wide_Value, Wide_Value, Value, Wide_Wide_Image, Wide_Image, and Image attributes of a floating point type to support special values such as infinities and NaNs.

56.1/3 An implementation may extend the Wide_Wide_Value, Wide_Value, and Value attributes of a character type to accept strings of the form “Hex_*hhhhhhhh*” (ignoring case) for any character (not just the ones for which Wide_Wide_Image would produce that form — see 3.5.2), as well as three-character strings of the form “‘*X*’”, where *X* is any character, including nongraphic characters.

Static Semantics

For a scalar type, the following language-defined representation aspect may be specified with an aspect_specification (see 13.1.1): 56.2/3

Default_Value 56.3/3

This aspect shall be specified by a static expression, and that expression shall be explicit, even if the aspect has a boolean type. Default_Value shall be specified only on a full_type_declaration.

If a derived type inherits a boolean Default_Value aspect, the aspect may be specified to have any value for the derived type. If a derived type *T* does not inherit a Default_Value aspect, it shall not specify such an aspect if it inherits a primitive subprogram that has a parameter of type *T* of mode **out**. 56.4/5

Name Resolution Rules

The expected type for the expression specified for the Default_Value aspect is the type defined by the full_type_declaration on which it appears. 56.5/3

NOTE 1 The evaluation of S'First or S'Last never raises an exception. If a scalar subtype S has a nonnull range, S'First and S'Last belong to this range. These values can, for example, always be assigned to a variable of subtype S. 57

NOTE 2 For a subtype of a scalar type, the result delivered by the attributes Succ, Pred, and Value can be outside to the subtype; similarly, the actual parameters of the attributes Succ, Pred, and Image can also be outside the subtype. 58/5

NOTE 3 For any value V (including any nongraphic character) of an enumeration subtype S without a specified Put_Image (see 4.10), S'Value(S'Image(V)) equals V, as do S'Wide_Value(S'Wide_Image(V)) and S'Wide_Wide_Value(S'Wide_Wide_Image(V)). None of these expressions ever raise Constraint_Error. 59/5

Examples

Examples of ranges: 60

```
-10 .. 10                                                    61
X .. X + 1
0.0 .. 2.0*Pi
Red .. Green      -- see 3.5.1
1 .. 0            -- a null range
Table'Range       -- a range attribute reference (see 3.6)
```

Examples of range constraints: 62

```
range -999.0 .. +999.0                                       63
range S'First+1 .. S'Last-1
```

3.5.1 Enumeration Types

An enumeration_type_definition defines an enumeration type. 1

Syntax

enumeration_type_definition ::= 2
  (enumeration_literal_specification {, enumeration_literal_specification})

enumeration_literal_specification ::= 3
  defining_identifier | defining_character_literal

defining_character_literal ::= character_literal 4

Legality Rules

The defining_identifiers in upper case and the defining_character_literals listed in an enumeration_type_definition shall be distinct. 5/3

Static Semantics

6/3 Each enumeration_literal_specification is the explicit declaration of the corresponding *enumeration literal*: it declares a parameterless function, whose defining name is the defining_identifier or defining_character_literal, and whose result subtype is the base subtype of the enumeration type.

7 Each enumeration literal corresponds to a distinct value of the enumeration type, and to a distinct position number. The position number of the value of the first listed enumeration literal is zero; the position number of the value of each subsequent enumeration literal is one more than that of its predecessor in the list.

8 The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.

9 If the same defining_identifier or defining_character_literal is specified in more than one enumeration_type_definition, the corresponding enumeration literals are said to be *overloaded*. At any place where an overloaded enumeration literal occurs in the text of a program, the type of the enumeration literal has to be determinable from the context (see 8.6).

Dynamic Semantics

10 The elaboration of an enumeration_type_definition creates the enumeration type and its first subtype, which is constrained to the base range of the type.

11 When called, the parameterless function associated with an enumeration literal returns the corresponding value of the enumeration type.

12 NOTE If an enumeration literal occurs in a context that does not otherwise suffice to determine the type of the literal, then qualification by the name of the enumeration type is one way to resolve the ambiguity (see 4.7).

Examples

13 *Examples of enumeration types and subtypes:*

14/5
```
type Day        is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
type Month_Name is (January, February, March, April, May, June, July,
                    August, September, October, November, December);
type Suit       is (Clubs, Diamonds, Hearts, Spades);
type Gender     is (M, F);
type Level      is (Low, Medium, Urgent);
type Color      is (White, Red, Yellow, Green, Blue, Brown, Black);
type Light      is (Red, Amber, Green); -- Red and Green are overloaded
```

15
```
type Hexa       is ('A', 'B', 'C', 'D', 'E', 'F');
type Mixed      is ('A', 'B', '*', B, None, '?', '%');
```

16
```
subtype Weekday is Day   range Mon .. Fri;
subtype Major   is Suit  range Hearts .. Spades;
subtype Rainbow is Color range Red .. Blue;
                                -- the Color Red, not the Light
```

3.5.2 Character Types

Static Semantics

1 An enumeration type is said to be a *character type* if at least one of its enumeration literals is a character_literal.

2/6 The predefined type Character is a character type having 256 values. Those values correspond to the 256 code points of Row 00 (also known as Latin-1) of the Basic Multilingual Plane (BMP) of The Unicode Standard, version 15.0.0. Each of the graphic characters of Row 00 of the BMP has a corresponding

character_literal in Character. Each of the nongraphic characters of Row 00 has a corresponding language-defined name, which is not usable as an enumeration literal, but which is usable with the attributes Image, Wide_Image, Wide_Wide_Image, Value, Wide_Value, and Wide_Wide_Value; these names are given in the definition of type Character in A.1, “The Package Standard”, but are set in *italics*.

The predefined type Wide_Character is a character type having 2**16 values. Those values correspond to the 2**16 code points of the Basic Multilingual Plane (BMP) of The Unicode Standard, version 15.0.0. Each of the graphic characters of the BMP has a corresponding character_literal in Wide_Character. The first 256 values of Wide_Character have the same character_literal or language-defined name as defined for Character. Each of the graphic_characters has a corresponding character_literal. 3/6

The predefined type Wide_Wide_Character is a character type having 2**32 values. The values with position numbers in the range 0 .. 16#10FFFF# correspond to the 17 * 2**16 code points of the codespace of The Unicode Standard, version 15.0.0. Each of the graphic_characters has a corresponding character_literal in Wide_Wide_Character. The first 2**16 values of Wide_Wide_Character have the same character_literal or language-defined name as defined for Wide_Character. 4/6

The characters whose code point is larger than 16#FF# and which are not graphic_characters have language-defined names which are formed by appending to the string "Hex_" the representation of their code point in hexadecimal as eight extended digits. As with other language-defined names, these names are usable only with the attributes (Wide_)Wide_Image and (Wide_)Wide_Value; they are not usable as enumeration literals. 5/3

Original Paragraphs 4 and 5 were deleted.

NOTE 1 The language-defined library package Characters.Latin_1 (see A.3.3) includes the declaration of constants denoting control characters, lower case characters, and special characters of the predefined type Character. 8

NOTE 2 A conventional character set such as *EBCDIC* can be declared as a character type; the internal codes of the characters can be specified by an enumeration_representation_clause as explained in 13.4. 9/5

Examples

Example of a character type: 10

```
type Roman_Digit is ('I', 'V', 'X', 'L', 'C', 'D', 'M');
```
11

3.5.3 Boolean Types

Static Semantics

There is a predefined enumeration type named Boolean, declared in the visible part of package Standard. It has the two enumeration literals False and True ordered with the relation False < True. Any descendant of the predefined type Boolean is called a *boolean* type. 1

3.5.4 Integer Types

An integer_type_definition defines an integer type; it defines either a *signed* integer type, or a *modular* integer type. The base range of a signed integer type includes at least the values of the specified range. A modular type is an integer type with all arithmetic modulo a specified positive *modulus*; such a type corresponds to an unsigned type with wrap-around semantics. 1

Syntax

integer_type_definition ::= 2
  signed_integer_type_definition | modular_type_definition

3 signed_integer_type_definition ::=
range *static*_simple_expression .. *static*_simple_expression

4 modular_type_definition ::= **mod** *static*_expression

Name Resolution Rules

5/5 Each simple_expression in a signed_integer_type_definition is expected to be of any integer type; they can be of different integer types. The expression in a modular_type_definition is likewise expected to be of any integer type.

Legality Rules

6 The simple_expressions of a signed_integer_type_definition shall be static, and their values shall be in the range System.Min_Int .. System.Max_Int.

7 The expression of a modular_type_definition shall be static, and its value (the *modulus*) shall be positive, and shall be no greater than System.Max_Binary_Modulus if a power of 2, or no greater than System.Max_Nonbinary_Modulus if not.

Static Semantics

8 The set of values for a signed integer type is the (infinite) set of mathematical integers, though only values of the base range of the type are fully supported for run-time operations. The set of values for a modular integer type are the values from 0 to one less than the modulus, inclusive.

9 A signed_integer_type_definition defines an integer type whose base range includes at least the values of the simple_expressions and is symmetric about zero, excepting possibly an extra negative value. A signed_integer_type_definition also defines a constrained first subtype of the type, with a range whose bounds are given by the values of the simple_expressions, converted to the type being defined.

10 A modular_type_definition defines a modular type whose base range is from zero to one less than the given modulus. A modular_type_definition also defines a constrained first subtype of the type with a range that is the same as the base range of the type.

11 There is a predefined signed integer subtype named Integer, declared in the visible part of package Standard. It is constrained to the base range of its type.

12 Integer has two predefined subtypes, declared in the visible part of package Standard:

13
```
subtype Natural  is Integer range 0 .. Integer'Last;
subtype Positive is Integer range 1 .. Integer'Last;
```

14 A type defined by an integer_type_definition is implicitly derived from *root_integer*, an anonymous predefined (specific) integer type, whose base range is System.Min_Int .. System.Max_Int. However, the base range of the new type is not inherited from *root_integer*, but is instead determined by the range or modulus specified by the integer_type_definition. Integer literals are all of the type *universal_integer*, the universal type (see 3.4.1) for the class rooted at *root_integer*, allowing their use with the operations of any integer type.

15 The *position number* of an integer value is equal to the value.

16/2 For every modular subtype S, the following attributes are defined:

16.1/2 S'Mod S'Mod denotes a function with the following specification:

16.2/2
```
function S'Mod (Arg : universal_integer)
  return S'Base
```

This function returns *Arg* **mod** S'Modulus, as a value of the type of S. 16.3/2

S'Modulus S'Modulus yields the modulus of the type of S, as a value of the type *universal_integer*. 17

Dynamic Semantics

The elaboration of an integer_type_definition creates the integer type and its first subtype. 18

For a modular type, if the result of the execution of a predefined operator (see 4.5) is outside the base range of the type, the result is reduced modulo the modulus of the type to a value that is within the base range of the type. 19

For a signed integer type, the exception Constraint_Error is raised by the execution of an operation that cannot deliver the correct result because it is outside the base range of the type. For any integer type, Constraint_Error is raised by the operators "/", "**rem**", and "**mod**" if the right operand is zero. 20

Implementation Requirements

In an implementation, the range of Integer shall include the range –2**15+1 .. +2**15–1. 21

If Long_Integer is predefined for an implementation, then its range shall include the range –2**31+1 .. +2**31–1. 22

System.Max_Binary_Modulus shall be at least 2**16. 23

Implementation Permissions

For the execution of a predefined operation of a signed integer type, it is optional to raise Constraint_Error if the result is outside the base range of the type, so long as the correct result is produced. 24/5

An implementation may provide additional predefined signed integer types, declared in the visible part of Standard, whose first subtypes have names of the form Short_Integer, Long_Integer, Short_Short_Integer, Long_Long_Integer, etc. Different predefined integer types are allowed to have the same base range. However, the range of Integer should be no wider than that of Long_Integer. Similarly, the range of Short_Integer (if provided) should be no wider than Integer. Corresponding recommendations apply to any other predefined integer types. An implementation may support base ranges for which there is no corresponding named integer type. The range of each first subtype should be the base range of its type. 25

An implementation may provide *nonstandard integer types*, descendants of *root_integer* that are declared outside of the specification of package Standard, which may have different characteristics than a type defined by an integer_type_definition. For example, a nonstandard integer type can have an asymmetric base range or it can be disallowed as an array or loop index (a very long integer). Any type descended from a nonstandard integer type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for “any integer type” are defined for a particular nonstandard integer type. In any case, such types are not permitted as explicit_generic_actual_parameters for formal scalar types — see 12.5.2. 26/5

For a one's complement machine, the high bound of the base range of a modular type whose modulus is one less than a power of 2 may be equal to the modulus, rather than one less than the modulus. It is implementation defined for which powers of 2, if any, this permission is exercised. 27

For a one's complement machine, implementations may support nonbinary modulus values greater than System.Max_Nonbinary_Modulus. It is implementation defined which specific values greater than System.Max_Nonbinary_Modulus, if any, are supported. 27.1/1

Implementation Advice

28 An implementation should support Long_Integer in addition to Integer if the target machine supports 32-bit (or longer) arithmetic. No other named integer subtypes are recommended for package Standard. Instead, appropriate named integer subtypes should be provided in the library package Interfaces (see B.2).

29 An implementation for a two's complement machine should support modular types with a binary modulus up to System.Max_Int*2+2. An implementation should support a nonbinary modulus up to Integer'Last.

30 NOTE 1 Integer literals are of the anonymous predefined integer type *universal_integer*. Other integer types have no literals. However, the overload resolution rules (see 8.6, "The Context of Overload Resolution") allow expressions of the type *universal_integer* whenever an integer type is expected.

31 NOTE 2 The same arithmetic operators are predefined for all signed integer types defined by a signed_integer_type_definition (see 4.5, "Operators and Expression Evaluation"). For modular types, these same operators are predefined, plus bit-wise logical operators (**and**, **or**, **xor**, and **not**). In addition, for the unsigned types declared in the language-defined package Interfaces (see B.2), functions are defined that provide bit-wise shifting and rotating.

32 NOTE 3 Modular types match a generic_formal_parameter_declaration of the form "**type** T **is mod** <>;"; signed integer types match "**type** T **is range** <>;" (see 12.5.2).

Examples

33 *Examples of integer types and subtypes:*

34
```
type Page_Num  is range 1 .. 2_000;
type Line_Size is range 1 .. Max_Line_Size;
```

35
```
subtype Small_Int   is Integer   range -10 .. 10;
subtype Column_Ptr  is Line_Size range 1 .. 10;
subtype Buffer_Size is Integer   range 0 .. Max;
```

36
```
type Byte        is mod 256; -- an unsigned byte
type Hash_Index  is mod 97;  -- modulus is prime
```

3.5.5 Operations of Discrete Types

Static Semantics

1 For every discrete subtype S, the following attributes are defined:

2 S'Pos S'Pos denotes a function with the following specification:

3
```
function S'Pos(Arg : S'Base)
  return universal_integer
```

4 This function returns the position number of the value of *Arg*, as a value of type *universal_integer*.

5 S'Val S'Val denotes a function with the following specification:

6
```
function S'Val(Arg : universal_integer)
  return S'Base
```

7 This function returns a value of the type of S whose position number equals the value of *Arg*. For the evaluation of a call on S'Val, if there is no value in the base range of its type with the given position number, Constraint_Error is raised.

7.1/4 For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S, the following attributes are defined:

7.2/4 S'First_Valid

S'First_Valid denotes the smallest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

S'Last_Valid 7.3/4

S'Last_Valid denotes the largest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S.

First_Valid and Last_Valid attribute_references are always static expressions. Any explicit predicate of S can only have been specified by a Static_Predicate aspect. 7.4/3

Implementation Advice

For the evaluation of a call on S'Pos for an enumeration subtype, if the value of the operand does not correspond to the internal code for any enumeration literal of its type (perhaps due to an uninitialized variable), then the implementation should raise Program_Error. This is particularly important for enumeration types with noncontiguous internal codes specified by an enumeration_representation_clause. 8

NOTE 1 Indexing and loop iteration use values of discrete types. 9

NOTE 2 The predefined operations of a discrete type include the assignment operation, qualification, the membership tests, and the relational operators; for a boolean type they include the short-circuit control forms and the logical operators; for an integer type they include type conversion to and from other numeric types, as well as the binary and unary adding operators – and +, the multiplying operators, the unary operator **abs**, and the exponentiation operator. The assignment operation is described in 5.2. The other predefined operations are described in Clause 4. 10

NOTE 3 As for all types, objects of a discrete type have Size and Address attributes (see 13.3). 11

NOTE 4 For a subtype of a discrete type, the result delivered by the attribute Val can be outside the subtype; similarly, the actual parameter of the attribute Pos can also be outside the subtype. The following relations are satisfied (in the absence of an exception) by these attributes: 12/5

```
S'Val(S'Pos(X)) = X
S'Pos(S'Val(N)) = N
```
13

Examples

Examples of attributes of discrete subtypes: 14

```
-- For the types and subtypes declared in 3.5.1 the following hold:
```
15/5

```
--  Color'First   = White,   Color'Last   = Black
--  Rainbow'First = Red,     Rainbow'Last = Blue
```
16

```
--  Color'Succ(Blue) = Rainbow'Succ(Blue) = Brown
--  Color'Pos(Blue)  = Rainbow'Pos(Blue)  = 4
--  Color'Val(0)     = Rainbow'Val(0)     = White
```
17

3.5.6 Real Types

Real types provide approximations to the real numbers, with relative bounds on errors for floating point types, and with absolute bounds for fixed point types. 1

Syntax

real_type_definition ::= 2
  floating_point_definition | fixed_point_definition

Static Semantics

A type defined by a real_type_definition is implicitly derived from *root_real*, an anonymous predefined (specific) real type. Hence, all real types, whether floating point or fixed point, are in the derivation class rooted at *root_real*. 3

Real literals are all of the type *universal_real*, the universal type (see 3.4.1) for the class rooted at *root_real*, allowing their use with the operations of any real type. Certain multiplying operators have a 4

result type of *universal_fixed* (see 4.5.5), the universal type for the class of fixed point types, allowing the result of the multiplication or division to be used where any specific fixed point type is expected.

Dynamic Semantics

The elaboration of a real_type_definition consists of the elaboration of the floating_point_definition or the fixed_point_definition.

Implementation Requirements

An implementation shall perform the run-time evaluation of a use of a predefined operator of *root_real* with an accuracy at least as great as that of any floating point type definable by a floating_point_definition.

Implementation Permissions

7/5 For the execution of a predefined operation of a real type, it is optional to raise Constraint_Error if the result is outside the base range of the type, so long as the correct result is produced, or the Machine_Overflows attribute of the type is False (see G.2.1).

8/5 An implementation may provide *nonstandard real types*, descendants of *root_real* that are declared outside of the specification of package Standard, which may have different characteristics than a type defined by a real_type_definition. For example, a nonstandard real type can have an asymmetric or unsigned base range, or its predefined operations can wrap around or "saturate" rather than overflow (modular or saturating arithmetic), or it can have a different accuracy model than is standard (see G.2.1). Any type descended from a nonstandard real type is also nonstandard. An implementation may place arbitrary restrictions on the use of such types; it is implementation defined whether operators that are predefined for "any real type" are defined for a particular nonstandard real type. In any case, such types are not permitted as explicit_generic_actual_parameters for formal scalar types — see 12.5.2.

NOTE As stated, real literals are of the anonymous predefined real type *universal_real*. Other real types have no literals. However, the overload resolution rules (see 8.6) allow expressions of the type *universal_real* whenever a real type is expected.

3.5.7 Floating Point Types

For floating point types, the error bound is specified as a relative precision by giving the required minimum number of significant decimal digits.

Syntax

floating_point_definition ::=
  digits *static_*expression [real_range_specification]

real_range_specification ::=
  range *static_*simple_expression .. *static_*simple_expression

Name Resolution Rules

The *requested decimal precision*, which is the minimum number of significant decimal digits required for the floating point type, is specified by the value of the expression given after the reserved word **digits**. This expression is expected to be of any integer type.

5/5 Each simple_expression of a real_range_specification is expected to be of any real type; the types can be different.

Legality Rules

The requested decimal precision shall be specified by a static expression whose value is positive and no greater than System.Max_Base_Digits. Each simple_expression of a real_range_specification shall also be static. If the real_range_specification is omitted, the requested decimal precision shall be no greater than System.Max_Digits. 6

A floating_point_definition is illegal if the implementation does not support a floating point type that satisfies the requested decimal precision and range. 7

Static Semantics

The set of values for a floating point type is the (infinite) set of rational numbers. The *machine numbers* of a floating point type are the values of the type that can be represented exactly in every unconstrained variable of the type. The base range (see 3.5) of a floating point type is symmetric around zero, except that it can include some extra negative values in some implementations. 8

The *base decimal precision* of a floating point type is the number of decimal digits of precision representable in objects of the type. The *safe range* of a floating point type is that part of its base range for which the accuracy corresponding to the base decimal precision is preserved by all predefined operations. 9

A floating_point_definition defines a floating point type whose base decimal precision is no less than the requested decimal precision. If a real_range_specification is given, the safe range of the floating point type (and hence, also its base range) includes at least the values of the simple expressions given in the real_range_specification. If a real_range_specification is not given, the safe (and base) range of the type includes at least the values of the range –10.0**(4*D) .. +10.0**(4*D) where D is the requested decimal precision. The safe range can include other values as well. The attributes Safe_First and Safe_Last give the actual bounds of the safe range. 10/5

A floating_point_definition also defines a first subtype of the type. If a real_range_specification is given, then the subtype is constrained to a range whose bounds are given by a conversion of the values of the simple_expressions of the real_range_specification to the type being defined. Otherwise, the subtype is unconstrained. 11

There is a predefined, unconstrained, floating point subtype named Float, declared in the visible part of package Standard. 12

Dynamic Semantics

The elaboration of a floating_point_definition creates the floating point type and its first subtype. 13

Implementation Requirements

In an implementation that supports floating point types with 6 or more digits of precision, the requested decimal precision for Float shall be at least 6. 14

If Long_Float is predefined for an implementation, then its requested decimal precision shall be at least 11. 15

Implementation Permissions

An implementation is allowed to provide additional predefined floating point types, declared in the visible part of Standard, whose (unconstrained) first subtypes have names of the form Short_Float, Long_Float, Short_Short_Float, Long_Long_Float, etc. Different predefined floating point types are allowed to have the same base decimal precision. However, the precision of Float should be no greater than that of Long_Float. Similarly, the precision of Short_Float (if provided) should be no greater than Float. 16/5

Corresponding recommendations apply to any other predefined floating point types. An implementation may support base decimal precisions for which there is no corresponding named floating point type.

Implementation Advice

17 An implementation should support Long_Float in addition to Float if the target machine supports 11 or more digits of precision. No other named floating point subtypes are recommended for package Standard. Instead, appropriate named floating point subtypes should be provided in the library package Interfaces (see B.2).

18 NOTE If a floating point subtype is unconstrained, then assignments to variables of the subtype involve only Overflow_Checks, never Range_Checks.

Examples

19 *Examples of floating point types and subtypes:*

20
```
type Coefficient is digits 10 range -1.0 .. 1.0;
```

21
```
type Real is digits 8;
type Mass is digits 7 range 0.0 .. 1.0E35;
```

22
```
subtype Probability is Real range 0.0 .. 1.0;
        -- a subtype with a smaller range
```

3.5.8 Operations of Floating Point Types

Static Semantics

1 The following attribute is defined for every floating point subtype S:

2/1 S'Digits S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type *T* is defined to be the largest value of *d* for which ceiling(*d* * log(10) / log(T'Machine_Radix)) + *g* <= T'Model_Mantissa where g is 0 if Machine_Radix is a positive power of 10 and 1 otherwise.

3 NOTE 1 The predefined operations of a floating point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the relational operators and the following predefined arithmetic operators: the binary and unary adding operators – and +, certain multiplying operators, the unary operator **abs**, and the exponentiation operator.

4 NOTE 2 As for all types, objects of a floating point type have Size and Address attributes (see 13.3). Other attributes of floating point types are defined in A.5.3.

3.5.9 Fixed Point Types

1 A fixed point type is either an ordinary fixed point type, or a decimal fixed point type. The error bound of a fixed point type is specified as an absolute value, called the *delta* of the fixed point type.

Syntax

2
```
fixed_point_definition ::=
   ordinary_fixed_point_definition | decimal_fixed_point_definition
```

3
```
ordinary_fixed_point_definition ::=
   delta static_expression  real_range_specification
```

4
```
decimal_fixed_point_definition ::=
   delta static_expression digits static_expression [real_range_specification]
```

digits_constraint ::= 5/4
  digits *static*_simple_expression [range_constraint]

Name Resolution Rules

For a type defined by a fixed_point_definition, the *delta* of the type is specified by the value of the expression given after the reserved word **delta**; this expression is expected to be of any real type. For a type defined by a decimal_fixed_point_definition (a *decimal* fixed point type), the number of significant decimal digits for its first subtype (the *digits* of the first subtype) is specified by the expression given after the reserved word **digits**; this expression is expected to be of any integer type. 6

The simple_expression of a digits_constraint is expected to be of any integer type. 6.1/4

Legality Rules

In a fixed_point_definition or digits_constraint, the expressions given after the reserved words **delta** and **digits** shall be static; their values shall be positive. 7

The set of values of a fixed point type comprise the integral multiples of a number called the *small* of the type. The *machine numbers* of a fixed point type are the values of the type that can be represented exactly in every unconstrained variable of the type. For a type defined by an ordinary_fixed_point_definition (an *ordinary* fixed point type), the *small* may be specified by an attribute_definition_clause (see 13.3); if so specified, it shall be no greater than the *delta* of the type. If not specified, the *small* of an ordinary fixed point type is an implementation-defined power of two less than or equal to the *delta*. 8/2

For a decimal fixed point type, the *small* equals the *delta*; the *delta* shall be a power of 10. If a real_range_specification is given, both bounds of the range shall be in the range –(10**_digits_–1)*_delta_ .. +(10**_digits_–1)*_delta_. 9

A fixed_point_definition is illegal if the implementation does not support a fixed point type with the given *small* and specified range or *digits*. 10

For a subtype_indication with a digits_constraint, the subtype_mark shall denote a decimal fixed point subtype. 11

Static Semantics

The base range (see 3.5) of a fixed point type is symmetric around zero, except possibly for an extra negative value in some implementations. 12

An ordinary_fixed_point_definition defines an ordinary fixed point type whose base range includes at least all multiples of *small* that are between the bounds specified in the real_range_specification. The base range of the type does not necessarily include the specified bounds themselves. An ordinary_fixed_point_definition also defines a constrained first subtype of the type, with each bound of its range given by the closer to zero of: 13

- the value of the conversion to the fixed point type of the corresponding expression of the real_range_specification; 14
- the corresponding bound of the base range. 15

A decimal_fixed_point_definition defines a decimal fixed point type whose base range includes at least the range –(10**_digits_–1)*_delta_ .. +(10**_digits_–1)*_delta_. A decimal_fixed_point_definition also defines a constrained first subtype of the type. If a real_range_specification is given, the bounds of the first subtype are given by a conversion of the values of the expressions of the real_range_specification. Otherwise, the range of the first subtype is –(10**_digits_–1)*_delta_ .. +(10**_digits_–1)*_delta_. 16

Dynamic Semantics

17 The elaboration of a fixed_point_definition creates the fixed point type and its first subtype.

18/4 For a digits_constraint on a decimal fixed point subtype with a given *delta*, if it does not have a range_constraint, then it specifies an implicit range –(10**D*–1)**delta* .. +(10**D*–1)**delta*, where *D* is the value of the simple_expression. A digits_constraint is *compatible* with a decimal fixed point subtype if the value of the simple_expression is no greater than the *digits* of the subtype, and if it specifies (explicitly or implicitly) a range that is compatible with the subtype.

19/4 The elaboration of a digits_constraint consists of the elaboration of the range_constraint, if any. If a range_constraint is given, a check is made that the bounds of the range are both in the range –(10**D*–1)**delta* .. +(10**D*–1)**delta*, where *D* is the value of the (static) simple_expression given after the reserved word **digits**. If this check fails, Constraint_Error is raised.

Implementation Requirements

20 The implementation shall support at least 24 bits of precision (including the sign bit) for fixed point types.

Implementation Permissions

21 Implementations are permitted to support only *small*s that are a power of two. In particular, all decimal fixed point type declarations can be disallowed. Note however that conformance with the Information Systems Annex requires support for decimal *small*s, and decimal fixed point type declarations with *digits* up to at least 18.

22/5 NOTE The specified bounds themselves can be outside the base range of an ordinary fixed point type so that the range specification can be given in a natural way, such as:

23
```
type Fraction is delta 2.0**(-15) range -1.0 .. 1.0;
```

24/5 With 2's complement hardware, such a type would typically have a signed 16-bit representation, using 1 bit for the sign and 15 bits for fraction, resulting in a base range of –1.0 .. 1.0–2.0**(–15).

Examples

25 *Examples of fixed point types and subtypes:*

26
```
type Volt is delta 0.125 range 0.0 .. 255.0;
```

27
```
   -- A pure fraction which requires all the available
   -- space in a word can be declared as the type Fraction:
type Fraction is delta System.Fine_Delta range -1.0 .. 1.0;
   -- Fraction'Last = 1.0 – System.Fine_Delta
```

28
```
type Money is delta 0.01 digits 15;  -- decimal fixed point
subtype Salary is Money digits 10;
   -- Money'Last = 10.0**13 – 0.01, Salary'Last = 10.0**8 – 0.01
```

3.5.10 Operations of Fixed Point Types

Static Semantics

1 The following attributes are defined for every fixed point subtype S:

2/5 S'Small — S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. Small may be specified for nonderived ordinary fixed point types via an attribute_definition_clause (see 13.3); the expression of such a clause shall be static and positive.

3 S'Delta — S'Delta denotes the *delta* of the fixed point subtype S. The value of this attribute is of the type *universal_real*.

S'Fore — S'Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type *universal_integer*.

S'Aft — S'Aft yields the number of decimal digits needed after the decimal point to accommodate the *delta* of the subtype S, unless the *delta* of the subtype S is greater than 0.1, in which case the attribute yields the value one. (S'Aft is the smallest positive integer N for which (10**N)*S'Delta is greater than or equal to one.) The value of this attribute is of the type *universal_integer*.

The following additional attributes are defined for every decimal fixed point subtype S:

S'Digits — S'Digits denotes the *digits* of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type *universal_integer*. Its value is determined as follows:

- For a first subtype or a subtype defined by a subtype_indication with a digits_constraint, the digits is the value of the expression given after the reserved word **digits**;
- For a subtype defined by a subtype_indication without a digits_constraint, the digits of the subtype is the same as that of the subtype denoted by the subtype_mark in the subtype_indication;
- The digits of a base subtype is the largest integer *D* such that the range –(10**_D_–1)*_delta_ .. +(10**_D_–1)*_delta_ is included in the base range of the type.

S'Scale — S'Scale denotes the *scale* of the subtype S, defined as the value N such that S'Delta = 10.0**(–N). The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S. The value of this attribute is of the type *universal_integer*.

S'Round — S'Round denotes a function with the following specification:

```
function S'Round(X : universal_real)
  return S'Base
```

The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S).

NOTE 1 All subtypes of a fixed point type will have the same value for the Delta attribute, in the absence of delta_constraints (see J.3).

NOTE 2 S'Scale is not always the same as S'Aft for a decimal subtype; for example, if S'Delta = 1.0 then S'Aft is 1 while S'Scale is 0.

NOTE 3 The predefined operations of a fixed point type include the assignment operation, qualification, the membership tests, and explicit conversion to and from other numeric types. They also include the relational operators and the following predefined arithmetic operators: the binary and unary adding operators – and +, multiplying operators, and the unary operator **abs**.

NOTE 4 As for all types, objects of a fixed point type have Size and Address attributes (see 13.3). Other attributes of fixed point types are defined in A.5.4.

3.6 Array Types

An *array* object is a composite object consisting of components which all have the same subtype. The name for a component of an array uses one or more index values belonging to specified discrete types. The value of an array object is a composite value consisting of the values of the components.

Syntax

2 array_type_definition ::=
unconstrained_array_definition | constrained_array_definition

3 unconstrained_array_definition ::=
array(index_subtype_definition {, index_subtype_definition}) **of** component_definition

4 index_subtype_definition ::= subtype_mark **range** <>

5 constrained_array_definition ::=
array (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition

6 discrete_subtype_definition ::= *discrete*_subtype_indication | range

7/2 component_definition ::=
[**aliased**] subtype_indication
| [**aliased**] access_definition

Name Resolution Rules

8 For a discrete_subtype_definition that is a range, the range shall resolve to be of some specific discrete type; which discrete type shall be determined without using any context other than the bounds of the range itself (plus the preference for *root_integer* — see 8.6).

Legality Rules

9 Each index_subtype_definition or discrete_subtype_definition in an array_type_definition defines an *index subtype*; its type (the *index type*) shall be discrete.

10 The subtype defined by the subtype_indication of a component_definition (the *component subtype*) shall be a definite subtype.

11/2 *This paragraph was deleted.*

Static Semantics

12 An array is characterized by the number of indices (the *dimensionality* of the array), the type and position of each index, the lower and upper bounds for each index, and the subtype of the components. The order of the indices is significant.

13 A one-dimensional array has a distinct component for each possible index value. A multidimensional array has a distinct component for each possible sequence of index values that can be formed by selecting one value for each index position (in the given order). The possible values for a given index are all the values between the lower and upper bounds, inclusive; this range of values is called the *index range*. The *bounds* of an array are the bounds of its index ranges. The *length* of a dimension of an array is the number of values of the index range of the dimension (zero for a null range). The *length* of a one-dimensional array is the length of its only dimension.

14 An array_type_definition defines an array type and its first subtype. For each object of this array type, the number of indices, the type and position of each index, and the subtype of the components are as in the type definition; the values of the lower and upper bounds for each index belong to the corresponding index subtype of its type, except for null arrays (see 3.6.1).

15/5 An unconstrained_array_definition defines an array type with an unconstrained first subtype. Each index_subtype_definition defines the corresponding index subtype to be the subtype denoted by the

subtype_mark. The compound delimiter <> (called a *box*) of an index_subtype_definition stands for an undefined range (different objects of the type can have different bounds).

A constrained_array_definition defines an array type with a constrained first subtype. Each discrete_subtype_definition defines the corresponding index subtype, as well as the corresponding index range for the constrained first subtype. The *constraint* of the first subtype consists of the bounds of the index ranges. 16

The discrete subtype defined by a discrete_subtype_definition is either that defined by the subtype_indication, or a subtype determined by the range as follows: 17

- If the type of the range resolves to *root_integer*, then the discrete_subtype_definition defines a subtype of the predefined type Integer with bounds given by a conversion to Integer of the bounds of the range; 18
- Otherwise, the discrete_subtype_definition defines a subtype of the type of the range, with the bounds given by the range. 19

The component_definition of an array_type_definition defines the nominal subtype of the components. If the reserved word **aliased** appears in the component_definition, then each component of the array is aliased (see 3.10). 20

Dynamic Semantics

The elaboration of an array_type_definition creates the array type and its first subtype, and consists of the elaboration of any discrete_subtype_definitions and the component_definition. 21

The elaboration of a discrete_subtype_definition that does not contain any per-object expressions creates the discrete subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. The elaboration of a discrete_subtype_definition that contains one or more per-object expressions is defined in 3.8. The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication or access_definition. The elaboration of any discrete_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order. 22/2

Static Semantics

For an array type with a scalar component type, the following language-defined representation aspect may be specified with an aspect_specification (see 13.1.1): 22.1/3

Default_Component_Value 22.2/3

This aspect shall be specified by a static expression, and that expression shall be explicit, even if the aspect has a boolean type. Default_Component_Value shall be specified only on a full_type_declaration.

If a derived type inherits a boolean Default_Component_Value aspect, the aspect may be specified to have any value for the derived type. 22.3/5

Name Resolution Rules

The expected type for the expression specified for the Default_Component_Value aspect is the component type of the array type defined by the full_type_declaration on which it appears. 22.4/3

NOTE 1 All components of an array have the same subtype. In particular, for an array of components that are one-dimensional arrays, this means that all components have the same bounds and hence the same length. 23

NOTE 2 Each elaboration of an array_type_definition creates a distinct array type. A consequence of this is that each object whose object_declaration contains an array_type_definition is of its own unique type. 24

Examples

25 *Examples of type declarations with unconstrained array definitions:*

26
```
type Vector     is array(Integer  range <>) of Real;
type Matrix     is array(Integer  range <>, Integer range <>) of Real;
type Bit_Vector is array(Integer  range <>) of Boolean;
type Roman      is array(Positive range <>) of Roman_Digit; -- see 3.5.2
```

27 *Examples of type declarations with constrained array definitions:*

28
```
type Table    is array(1 .. 10) of Integer;
type Schedule is array(Day) of Boolean;
type Line     is array(1 .. Max_Line_Size) of Character;
```

29 *Examples of object declarations with array type definitions:*

30/2
```
Grid      : array(1 .. 80, 1 .. 100) of Boolean;
Mix       : array(Color range Red .. Green) of Boolean;
Msg_Table : constant array(Error_Code) of access constant String :=
      (Too_Big => new String'("Result too big"), Too_Small => ...);
Page      : array(Positive range <>) of Line := -- an array of arrays
  (1 | 50  => Line'(1 | Line'Last => '+', others => '-'), -- see 4.3.3
   2 .. 49 => Line'(1 | Line'Last => '|', others => ' '));
    -- Page is constrained by its initial value to (1..50)
```

3.6.1 Index Constraints and Discrete Ranges

1 An index_constraint determines the range of possible values for every index of an array subtype, and thereby the corresponding array bounds.

Syntax

2 index_constraint ::= (discrete_range {, discrete_range})

3 discrete_range ::= *discrete*_subtype_indication | range

Name Resolution Rules

4 The type of a discrete_range is the type of the subtype defined by the subtype_indication, or the type of the range. For an index_constraint, each discrete_range shall resolve to be of the type of the corresponding index.

Legality Rules

5 An index_constraint shall appear only in a subtype_indication whose subtype_mark denotes either an unconstrained array subtype, or an unconstrained access subtype whose designated subtype is an unconstrained array subtype; in either case, the index_constraint shall provide a discrete_range for each index of the array type.

Static Semantics

6 A discrete_range defines a range whose bounds are given by the range, or by the range of the subtype defined by the subtype_indication.

Dynamic Semantics

7 An index_constraint is *compatible* with an unconstrained array subtype if and only if the index range defined by each discrete_range is compatible (see 3.5) with the corresponding index subtype. If any of the discrete_ranges defines a null range, any array thus constrained is a *null array*, having no components. An array value *satisfies* an index_constraint if at each index position the array value and the index_constraint have the same index bounds.

The elaboration of an index_constraint consists of the evaluation of the discrete_range(s), in an arbitrary order. The evaluation of a discrete_range consists of the elaboration of the subtype_indication or the evaluation of the range. 8

When the *upper bound is calculated* from the lower bound and another value representing the length of an array, the following steps are performed: 9/6

1. determine the position number of the lower bound, giving position number *L*; 10/6
2. add one less than the length value to *L*, giving the position number *U*; then 11/6
3. the value of the index type whose position number is *U* is the value of the upper bound; if there is no value of the index type whose position number is *U*, Constraint_Error is raised. 12/6

NOTE 1 The elaboration of a subtype_indication consisting of a subtype_mark followed by an index_constraint checks the compatibility of the index_constraint with the subtype_mark (see 3.2.2). 13

NOTE 2 Even if an array value does not satisfy the index constraint of an array subtype, Constraint_Error is not raised on conversion to the array subtype, so long as the length of each dimension of the array value and the array subtype match. See 4.6. 14

Examples

Examples of array declarations including an index constraint: 15

```
Board     : Matrix(1 .. 8,  1 .. 8);   -- see 3.6
Rectangle : Matrix(1 .. 20, 1 .. 30);
Inverse   : Matrix(1 .. N,  1 .. N);   -- N can be nonstatic
```
16/5

```
Filter    : Bit_Vector(0 .. 31);       -- see 3.6
```
17/5

Example of array declaration with a constrained array subtype: 18

```
My_Schedule : Schedule;   -- all arrays of type Schedule have the same bounds
```
19

Example of record type with a component that is an array: 20

```
type Var_Line(Length : Natural) is
   record
      Image : String(1 .. Length);
   end record;
Null_Line : Var_Line(0);   -- Null_Line.Image is a null array
```
21 22

3.6.2 Operations of Array Types

Legality Rules

The argument N used in the attribute_designators for the N-th dimension of an array shall be a static expression of some integer type. The value of N shall be positive (nonzero) and no greater than the dimensionality of the array. 1

Static Semantics

The following attributes are defined for a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 2/1

A'First — A'First denotes the lower bound of the first index range; its type is the corresponding index type. 3

A'First(N) — A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type. 4

A'Last — A'Last denotes the upper bound of the first index range; its type is the corresponding index type. 5

6 A'Last(N) A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type.

7 A'Range A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once.

8 A'Range(N) A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once.

9 A'Length A'Length denotes the number of values of the first index range (zero for a null range); its type is *universal_integer*.

10 A'Length(N) A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is *universal_integer*.

Implementation Advice

11/3 An implementation should normally represent multidimensional arrays in row-major order, consistent with the notation used for multidimensional array aggregates (see 4.3.3). However, if convention Fortran is specified for a multidimensional array type, then column-major order should be used instead (see B.5, “Interfacing with Fortran”).

12 NOTE 1 The attribute_references A'First and A'First(1) denote the same value. A similar relation exists for the attribute_references A'Last, A'Range, and A'Length. The following relation is satisfied (except for a null array) by the above attributes if the index type is an integer type:

13
```
A'Length(N) = A'Last(N) - A'First(N) + 1
```

14 NOTE 2 An array type is limited if its component type is limited (see 7.5).

15 NOTE 3 The predefined operations of an array type include the membership tests, qualification, and explicit conversion. If the array type is not limited, they also include assignment and the predefined equality operators. For a one-dimensional array type, they include the predefined concatenation operators (if nonlimited) and, if the component type is discrete, the predefined relational operators; if the component type is boolean, the predefined logical operators are also included.

16/2 NOTE 4 A component of an array can be named with an indexed_component. A value of an array type can be specified with an array_aggregate. For a one-dimensional array type, a slice of the array can be named; also, string literals are defined if the component type is a character type.

Examples

17/5 *Examples (using arrays declared in the examples of 3.6.1):*

18
```
-- Filter'First      =  0   Filter'Last       = 31   Filter'Length = 32
-- Rectangle'Last(1) = 20   Rectangle'Last(2) = 30
```

3.6.3 String Types

Static Semantics

1 A one-dimensional array type whose component type is a character type is called a *string type*.

2/2 There are three predefined string types, String, Wide_String, and Wide_Wide_String, each indexed by values of the predefined subtype Positive; these are declared in the visible part of package Standard:

3
```
subtype Positive is Integer range 1 .. Integer'Last;
```

4/2
```
type String is array(Positive range <>) of Character;
type Wide_String is array(Positive range <>) of Wide_Character;
type Wide_Wide_String is array(Positive range <>) of Wide_Wide_Character;
```

5 NOTE String literals (see 2.6 and 4.2) are defined for all string types. The concatenation operator & is predefined for string types, as for all nonlimited one-dimensional array types. The ordering operators <, <=, >, and >= are predefined for

string types, as for all one-dimensional discrete array types; these ordering operators correspond to lexicographic order (see 4.5.2).

Examples

Examples of string objects: 6

```
Stars      : String(1 .. 120) := (1 .. 120 => '*' );                    7
Question   : constant String  := "How many characters?";
                      -- Question'First = 1, Question'Last = 20
                      -- Question'Length = 20 (the number of characters)

Ask_Twice  : String  := Question & Question; -- constrained to (1..40)  8
Ninety_Six : constant Roman   := "XCVI";     -- see 3.5.2 and 3.6
```

3.7 Discriminants

A composite type (other than an array or interface type) can have discriminants, which parameterize the type. A known_discriminant_part specifies the discriminants of a composite type. A discriminant of an object is a component of the object, and is either of a discrete type or an access type. An unknown_discriminant_part in the declaration of a view of a type specifies that the discriminants of the type are unknown for the given view; all subtypes of such a view are indefinite subtypes. 1/2

Syntax

discriminant_part ::= unknown_discriminant_part | known_discriminant_part 2/2

unknown_discriminant_part ::= (<>) 3

known_discriminant_part ::= 4
  (discriminant_specification {; discriminant_specification})

discriminant_specification ::= 5/5
  defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
    [aspect_specification]
| defining_identifier_list : access_definition [:= default_expression]
    [aspect_specification]

default_expression ::= expression 6

Name Resolution Rules

The expected type for the default_expression of a discriminant_specification is that of the corresponding discriminant. 7

Legality Rules

A discriminant_part is only permitted in a declaration for a composite type that is not an array or interface type (this includes generic formal types). A type declared with a known_discriminant_part is called a *discriminated* type, as is a type that inherits (known) discriminants. 8/2

The subtype of a discriminant may be defined by an optional null_exclusion and a subtype_mark, in which case the subtype_mark shall denote a discrete or access subtype, or it may be defined by an access_definition. A discriminant that is defined by an access_definition is called an *access discriminant* and is of an anonymous access type. 9/2

9.1/3 Default_expressions shall be provided either for all or for none of the discriminants of a known_discriminant_part. No default_expressions are permitted in a known_discriminant_part in a declaration of a nonlimited tagged type or a generic formal type.

10/3 A discriminant_specification for an access discriminant may have a default_expression only in the declaration for an immutably limited type (see 7.5). In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

11/2 *This paragraph was deleted.*

12 For a type defined by a derived_type_definition, if a known_discriminant_part is provided in its declaration, then:

13 • The parent subtype shall be constrained;

14 • If the parent type is not a tagged type, then each discriminant of the derived type shall be used in the constraint defining the parent subtype;

15 • If a discriminant is used in the constraint defining the parent subtype, the subtype of the discriminant shall be statically compatible (see 4.9.1) with the subtype of the corresponding parent discriminant.

16/3 *This paragraph was deleted.*

Static Semantics

17 A discriminant_specification declares a discriminant; the subtype_mark denotes its subtype unless it is an access discriminant, in which case the discriminant's subtype is the anonymous access-to-variable subtype defined by the access_definition.

18 For a type defined by a derived_type_definition, each discriminant of the parent type is either inherited, constrained to equal some new discriminant of the derived type, or constrained to the value of an expression. When inherited or constrained to equal some new discriminant, the parent discriminant and the discriminant of the derived type are said to *correspond*. Two discriminants also correspond if there is some common discriminant to which they both correspond. A discriminant corresponds to itself as well. If a discriminant of a parent type is constrained to a specific value by a derived_type_definition, then that discriminant is said to be *specified* by that derived_type_definition.

19 A constraint that appears within the definition of a discriminated type *depends on a discriminant* of the type if it names the discriminant as a bound or discriminant value. A component_definition depends on a discriminant if its constraint depends on the discriminant, or on a discriminant that corresponds to it.

20 A component *depends on a discriminant* if:

21 • Its component_definition depends on the discriminant; or

22 • It is declared in a variant_part that is governed by the discriminant; or

23 • It is a component inherited as part of a derived_type_definition, and the constraint of the *parent*_subtype_indication depends on the discriminant; or

24 • It is a subcomponent of a component that depends on the discriminant.

25 Each value of a discriminated type includes a value for each component of the type that does not depend on a discriminant; this includes the discriminants themselves. The values of discriminants determine which other component values are present in the value of the discriminated type.

26 A type declared with a known_discriminant_part is said to have *known discriminants*; its first subtype is unconstrained. A type declared with an unknown_discriminant_part is said to have *unknown*

discriminants. A type declared without a discriminant_part has no discriminants, unless it is a derived type; if derived, such a type has the same sort of discriminants (known, unknown, or none) as its parent (or ancestor) type. A tagged class-wide type also has unknown discriminants. Any subtype of a type with unknown discriminants is an unconstrained and indefinite subtype (see 3.2 and 3.3).

Dynamic Semantics

For an access discriminant, its access_definition is elaborated when the value of the access discriminant is defined: by evaluation of its default_expression, by elaboration of a discriminant_constraint, or by an assignment that initializes the enclosing object. 27/2

NOTE 1 If a discriminated type has default_expressions for its discriminants, then unconstrained variables of the type are permitted, and the values of the discriminants can be changed by an assignment to such a variable. If defaults are not provided for the discriminants, then all variables of the type are constrained, either by explicit constraint or by their initial value; the values of the discriminants of such a variable cannot be changed after initialization. 28

NOTE 2 The default_expression for a discriminant of a type is evaluated when an object of an unconstrained subtype of the type is created. 29

NOTE 3 Assignment to a discriminant of an object (after its initialization) is not allowed, since the name of a discriminant is a constant; neither assignment_statements nor assignments inherent in passing as an **in out** or **out** parameter are allowed. Note however that the value of a discriminant can be changed by assigning to the enclosing object, presuming it is an unconstrained variable. 30

NOTE 4 A discriminant that is of a named access type is not called an access discriminant; that term is used only for discriminants defined by an access_definition. 31

Examples

Examples of discriminated types: 32

```
type Buffer(Size : Buffer_Size := 100)  is          -- see 3.5.4
   record
      Pos   : Buffer_Size := 0;
      Value : String(1 .. Size);
   end record;
```
33

```
type Matrix_Rec(Rows, Columns : Integer) is
   record
      Mat : Matrix(1 .. Rows, 1 .. Columns);        -- see 3.6
   end record;
```
34

```
type Square(Side : Integer) is new
   Matrix_Rec(Rows => Side, Columns => Side);
```
35

```
type Double_Square(Number : Integer) is
   record
      Left  : Square(Number);
      Right : Square(Number);
   end record;
```
36

```
task type Worker(Prio : System.Priority; Buf : access Buffer)
   with Priority => Prio is -- see D.1
   -- discriminants used to parameterize the task type (see 9.1)
   entry Fill;
   entry Drain;
end Worker;
```
37/3

3.7.1 Discriminant Constraints

1 A discriminant_constraint specifies the values of the discriminants for a given discriminated type.

Syntax

2 discriminant_constraint ::=
(discriminant_association {, discriminant_association})

3/5 discriminant_association ::=
[*discriminant*_selector_name {'|' *discriminant*_selector_name} =>] expression

4 A discriminant_association is said to be *named* if it has one or more *discriminant*_selector_names; it is otherwise said to be *positional*. In a discriminant_constraint, any positional associations shall precede any named associations.

Name Resolution Rules

5 Each selector_name of a named discriminant_association shall resolve to denote a discriminant of the subtype being constrained; the discriminants so named are the *associated discriminants* of the named association. For a positional association, the *associated discriminant* is the one whose discriminant_specification occurred in the corresponding position in the known_discriminant_part that defined the discriminants of the subtype being constrained.

6 The expected type for the expression in a discriminant_association is that of the associated discriminant(s).

Legality Rules

7/3 A discriminant_constraint is only allowed in a subtype_indication whose subtype_mark denotes either an unconstrained discriminated subtype, or an unconstrained access subtype whose designated subtype is an unconstrained discriminated subtype. However, in the case of an access subtype, a discriminant_constraint is legal only if any dereference of a value of the access type is known to be constrained (see 3.3). In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

8 A named discriminant_association with more than one selector_name is allowed only if the named discriminants are all of the same type. A discriminant_constraint shall provide exactly one value for each discriminant of the subtype being constrained.

9/3 *This paragraph was deleted.*

Dynamic Semantics

10 A discriminant_constraint is *compatible* with an unconstrained discriminated subtype if each discriminant value belongs to the subtype of the corresponding discriminant.

11 A composite value *satisfies* a discriminant constraint if and only if each discriminant of the composite value has the value imposed by the discriminant constraint.

12/5 For the elaboration of a discriminant_constraint, the expressions in the discriminant_associations are evaluated in an arbitrary order and converted to the type of the associated discriminant (which can raise Constraint_Error — see 4.6); the expression of a named association is evaluated (and converted) once for each associated discriminant. The result of each evaluation and conversion is the value imposed by the constraint for the associated discriminant.

NOTE The rules of the language ensure that a discriminant of an object always has a value, either from explicit or implicit initialization. 13

Examples

Examples (using types declared above in 3.7): 14/5

```
Large   : Buffer(200);  -- constrained, always 200 characters                    15
                        --  (explicit discriminant value)
Message : Buffer;       -- unconstrained, initially 100 characters
                        --  (default discriminant value)
Basis   : Square(5);    -- constrained, always 5 by 5
Illegal : Square;       -- illegal, a Square has to be constrained
```

3.7.2 Operations of Discriminated Types

If a discriminated type has default_expressions for its discriminants, then unconstrained variables of the type are permitted, and the discriminants of such a variable can be changed by assignment to the variable. For a formal parameter of such a type, an attribute is provided to determine whether the corresponding actual parameter is constrained or unconstrained. 1

Static Semantics

For a prefix A that is of a discriminated type (after any implicit dereference), the following attribute is defined: 2

A'Constrained 3/5

Yields the value True if A denotes a constant, a value, a tagged object, or a constrained variable, and False otherwise. The value of this attribute is of the predefined type Boolean.

Erroneous Execution

The execution of a construct is erroneous if the construct has a constituent that is a name denoting a subcomponent that depends on discriminants, and the value of any of these discriminants is changed by this execution between evaluating the name and the last use (within this execution) of the subcomponent denoted by the name. 4

3.8 Record Types

A record object is a composite object consisting of named components. The value of a record object is a composite value consisting of the values of the components. 1

Syntax

record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition 2

record_definition ::= 3/5

```
   record
      component_list
   end record [record_identifier]
 | null record
```

component_list ::= 4

```
     component_item {component_item}
  | {component_item} variant_part
  | null;
```

5/1 component_item ::= component_declaration | aspect_clause

6/3 component_declaration ::=
defining_identifier_list : component_definition [:= default_expression]
[aspect_specification];

6.1/5 If a *record_*identifier appears at the end of the record_definition, it shall repeat the defining_identifier of the enclosing full_type_declaration.

Name Resolution Rules

7 The expected type for the default_expression, if any, in a component_declaration is the type of the component.

Legality Rules

8/2 *This paragraph was deleted.*

9/2 Each component_declaration declares a component of the record type. Besides components declared by component_declarations, the components of a record type include any components declared by discriminant_specifications of the record type declaration. The identifiers of all components of a record type shall be distinct.

10 Within a type_declaration, a name that denotes a component, protected subprogram, or entry of the type is allowed only in the following cases:

11/3 - A name that denotes any component, protected subprogram, or entry is allowed within an aspect_specification, an operational item, or a representation item that occurs within the declaration of the composite type.

12/3 - A name that denotes a noninherited discriminant is allowed within the declaration of the type, but not within the discriminant_part. If the discriminant is used to define the constraint of a component, the bounds of an entry family, or the constraint of the parent subtype in a derived_type_definition, then its name shall appear alone as a direct_name (not as part of a larger expression or expanded name). A discriminant shall not be used to define the constraint of a scalar component.

13 If the name of the current instance of a type (see 8.6) is used to define the constraint of a component, then it shall appear as a direct_name that is the prefix of an attribute_reference whose result is of an access type, and the attribute_reference shall appear alone.

Static Semantics

13.1/3 If a record_type_definition includes the reserved word **limited**, the type is called an *explicitly limited record* type.

14 The component_definition of a component_declaration defines the (nominal) subtype of the component. If the reserved word **aliased** appears in the component_definition, then the component is aliased (see 3.10).

15 If the component_list of a record type is defined by the reserved word **null** and there are no discriminants, then the record type has no components and all records of the type are *null records*. A record_definition of **null record** is equivalent to **record null; end record**.

Dynamic Semantics

The elaboration of a record_type_definition creates the record type and its first subtype, and consists of the elaboration of the record_definition. The elaboration of a record_definition consists of the elaboration of its component_list, if any. 16

The elaboration of a component_list consists of the elaboration of the component_items and variant_part, if any, in the order in which they appear. The elaboration of a component_declaration consists of the elaboration of the component_definition. 17

Within the definition of a composite type, if a component_definition of a component_declaration or the discrete_subtype_definition of an entry_declaration for an entry family (see 9.5.2) includes a name that denotes a discriminant of the type, or that is an attribute_reference whose prefix denotes the current instance of the type, the expression containing the name is called a *per-object expression*, and the constraint or range being defined is called a *per-object constraint*. For the elaboration of a component_definition of a component_declaration or the discrete_subtype_definition of an entry_declaration for an entry family, if the component subtype is defined by an access_definition or if the constraint or range of the subtype_indication or discrete_subtype_definition is not a per-object constraint, then the access_definition, subtype_indication, or discrete_subtype_definition is elaborated. On the other hand, if the constraint or range is a per-object constraint, then the elaboration consists of the evaluation of any included expression that is not part of a per-object expression. Each such expression is evaluated once unless it is part of a named association in a discriminant constraint, in which case it is evaluated once for each associated discriminant. 18/6

When a per-object constraint is elaborated (as part of creating an object), each per-object expression of the constraint is evaluated. For other expressions, the values determined during the elaboration of the component_definition or entry_declaration are used. Any checks associated with the enclosing subtype_indication or discrete_subtype_definition are performed, including the subtype compatibility check (see 3.2.2), and the associated subtype is created. 18.1/1

NOTE 1 A component_declaration with several identifiers is equivalent to a sequence of single component_declarations, as explained in 3.3.1. 19

NOTE 2 The default_expression of a record component is only evaluated upon the creation of a default-initialized object of the record type (presuming the object has the component, if it is in a variant_part — see 3.3.1). 20

NOTE 3 The subtype defined by a component_definition (see 3.6) has to be a definite subtype. 21

NOTE 4 If a record type does not have a variant_part, then the same components are present in all values of the type. 22

NOTE 5 A record type is limited if it has the reserved word **limited** in its definition, or if any of its components are limited (see 7.5). 23

NOTE 6 The predefined operations of a record type include membership tests, qualification, and explicit conversion. If the record type is nonlimited, they also include assignment and the predefined equality operators. 24

NOTE 7 A component of a record can be named with a selected_component. A value of a record can be specified with a record_aggregate. 25/2

Examples

Examples of record type declarations: 26

```
type Date is                                                    27/5
   record
      Day   : Integer range 1 .. 31;
      Month : Month_Name;                    -- see 3.5.1
      Year  : Integer range 0 .. 4000;
   end record;
```

28/5
```
type Complex is
   record
      Re : Real := 0.0;
      Im : Real := 0.0;
   end record Complex;
```

29 *Examples of record variables:*

30
```
Tomorrow, Yesterday : Date;
A, B, C : Complex;
```

31
```
-- both components of A, B, and C are implicitly initialized to zero
```

3.8.1 Variant Parts and Discrete Choices

1 A record type with a variant_part specifies alternative lists of components. Each variant defines the components for the value or values of the discriminant covered by its discrete_choice_list.

Syntax

2
```
variant_part ::=
  case discriminant_direct_name is
     variant
    {variant}
  end case;
```

3
```
variant ::=
  when discrete_choice_list =>
    component_list
```

4/5
```
discrete_choice_list ::= discrete_choice {'|' discrete_choice}
```

5/3
```
discrete_choice ::=
  choice_expression | discrete_subtype_indication | range | others
```

Name Resolution Rules

6 The discriminant_direct_name shall resolve to denote a discriminant (called the *discriminant of the variant_part*) specified in the known_discriminant_part of the full_type_declaration that contains the variant_part. The expected type for each discrete_choice in a variant is the type of the discriminant of the variant_part.

Legality Rules

7 The discriminant of the variant_part shall be of a discrete type.

8/3 The choice_expressions, subtype_indications, and ranges given as discrete_choices in a variant_part shall be static. The discrete_choice **others** shall appear alone in a discrete_choice_list, and such a discrete_choice_list, if it appears, shall be the last one in the enclosing construct.

9 A discrete_choice is defined to *cover a value* in the following cases:

10/3 - A discrete_choice that is a choice_expression covers a value if the value equals the value of the choice_expression converted to the expected type.

10.1/4 - A discrete_choice that is a subtype_indication covers all values (possibly none) that belong to the subtype and that satisfy the static predicates of the subtype (see 3.2.4).

11/3 - A discrete_choice that is a range covers all values (possibly none) that belong to the range.

- The discrete_choice **others** covers all values of its expected type that are not covered by previous discrete_choice_lists of the same construct. 12

A discrete_choice_list covers a value if one of its discrete_choices covers the value. 13

The possible values of the discriminant of a variant_part shall be covered as follows: 14

- If the discriminant is of a static constrained scalar subtype then, except within an instance of a generic unit, each non-**others** discrete_choice shall cover only values in that subtype that satisfy its predicates, and each value of that subtype that satisfies its predicates shall be covered by some discrete_choice (either explicitly or by **others**); 15/4
- If the type of the discriminant is a descendant of a generic formal scalar type, then the variant_part shall have an **others** discrete_choice; 16/3
- Otherwise, each value of the base range of the type of the discriminant shall be covered (either explicitly or by **others**). 17

Two distinct discrete_choices of a variant_part shall not cover the same value. 18

Static Semantics

If the component_list of a variant is specified by **null**, the variant has no components. 19

The discriminant of a variant_part is said to *govern* the variant_part and its variants. In addition, the discriminant of a derived type governs a variant_part and its variants if it corresponds (see 3.7) to the discriminant of the variant_part. 20

Dynamic Semantics

A record value contains the values of the components of a particular variant only if the value of the discriminant governing the variant is covered by the discrete_choice_list of the variant. This rule applies in turn to any further variant that is, itself, included in the component_list of the given variant. 21

When an object of a discriminated type *T* is initialized by default, Constraint_Error is raised if no discrete_choice_list of any variant of a variant_part of *T* covers the value of the discriminant that governs the variant_part. When a variant_part appears in the component_list of another variant *V*, this test is only applied if the value of the discriminant governing *V* is covered by the discrete_choice_list of *V*. 21.1/3

The elaboration of a variant_part consists of the elaboration of the component_list of each variant in the order in which they appear. 22

Examples

Example of record type with a variant part: 23

```
type Device is (Printer, Disk, Drum);                                    24
type State  is (Open, Closed);

type Peripheral(Unit : Device := Disk) is                                25
   record
      Status : State;
      case Unit is
         when Printer =>
            Line_Count : Integer range 1 .. Page_Size;
         when others =>
            Cylinder   : Cylinder_Index;
            Track      : Track_Number;
      end case;
   end record;
```

26 *Examples of record subtypes:*

27
```
subtype Drum_Unit is Peripheral(Drum);
subtype Disk_Unit is Peripheral(Disk);
```

28 *Examples of constrained record variables:*

29
```
Writer   : Peripheral(Unit  => Printer);
Archive  : Disk_Unit;
```

3.9 Tagged Types and Type Extensions

1 Tagged types and type extensions support object-oriented programming, based on inheritance with extension and run-time polymorphism via *dispatching operations*.

Static Semantics

2/2 A record type or private type that has the reserved word **tagged** in its declaration is called a *tagged* type. In addition, an interface type is a tagged type, as is a task or protected type derived from an interface (see 3.9.4). When deriving from a tagged type, as for any derived type, additional primitive subprograms may be defined, and inherited primitive subprograms may be overridden. The derived type is called an *extension* of its ancestor types, or simply a *type extension*.

2.1/2 Every type extension is also a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a noninterface synchronized tagged type (see 3.9.4). A record extension is defined by a derived_type_definition with a record_extension_part (see 3.9.1), which may include the definition of additional components. A private extension, which is a partial view of a record extension or of a synchronized tagged type, can be declared in the visible part of a package (see 7.3) or in a generic formal part (see 12.5.1).

3 An object of a tagged type has an associated (run-time) *tag* that identifies the specific tagged type used to create the object originally. The tag of an operand of a class-wide tagged type *T*'Class controls which subprogram body is to be executed when a primitive subprogram of type *T* is applied to the operand (see 3.9.2); using a tag to control which body to execute is called *dispatching*.

4/2 The tag of a specific tagged type identifies the full_type_declaration of the type, and for a type extension, is sufficient to uniquely identify the type among all descendants of the same ancestor. If a declaration for a tagged type occurs within a generic_package_declaration, then the corresponding type declarations in distinct instances of the generic package are associated with distinct tags. For a tagged type that is local to a generic package body and with all of its ancestors (if any) also local to the generic body, the language does not specify whether repeated instantiations of the generic body result in distinct tags.

5 The following language-defined library package exists:

6/5
```
package Ada.Tags
    with Preelaborate, Nonblocking, Global => in out synchronized is
    type Tag is private
       with Preelaborable_Initialization;
```
6.1/2
```
    No_Tag : constant Tag;
```
7/2
```
    function Expanded_Name(T : Tag) return String;
    function Wide_Expanded_Name(T : Tag) return Wide_String;
    function Wide_Wide_Expanded_Name(T : Tag) return Wide_Wide_String;
    function External_Tag(T : Tag) return String;
    function Internal_Tag(External : String) return Tag;
```

```
      function Descendant_Tag(External : String;                                    7.1/2
                              Ancestor : Tag) return Tag;
      function Is_Descendant_At_Same_Level(Descendant, Ancestor : Tag)
          return Boolean;
      function Parent_Tag (T : Tag) return Tag;                                     7.2/2
      type Tag_Array is array (Positive range <>) of Tag;                           7.3/2
      function Interface_Ancestor_Tags (T : Tag) return Tag_Array;                  7.4/2
      function Is_Abstract (T : Tag) return Boolean;                                7.5/3
      Tag_Error : exception;                                                        8
   private                                                                          9
      ... -- not specified by the language
   end Ada.Tags;
```

No_Tag is the default initial value of type Tag. 9.1/2

The function Wide_Wide_Expanded_Name returns the full expanded name of the first subtype of the specific type identified by the tag, in upper case, starting with a root library unit. The result is implementation defined if the type is declared within an unnamed block_statement. 10/2

The function Expanded_Name (respectively, Wide_Expanded_Name) returns the same sequence of graphic characters as that defined for Wide_Wide_Expanded_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Expanded_Name for the same value of the argument. 10.1/2

The function External_Tag returns a string to be used in an external representation for the given tag. The call External_Tag(S'Tag) is equivalent to the attribute_reference S'External_Tag (see 13.3). 11

The string returned by the functions Expanded_Name, Wide_Expanded_Name, Wide_Wide_Expanded_Name, and External_Tag has lower bound 1. 11.1/2

The function Internal_Tag returns a tag that corresponds to the given external tag, or raises Tag_Error if the given string is not the external tag for any specific type of the partition. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created (see 13.14). 12/2

The function Descendant_Tag returns the (internal) tag for the type that corresponds to the given external tag and is both a descendant of the type identified by the Ancestor tag and has the same accessibility level as the identified ancestor. Tag_Error is raised if External is not the external tag for such a type. Tag_Error is also raised if the specific type identified is a library-level type whose tag has not yet been created, or if the given external tag identifies more than one type that has the appropriate Ancestor and accessibility level. 12.1/3

The function Is_Descendant_At_Same_Level returns True if the Descendant tag identifies a type that is both a descendant of the type identified by Ancestor and at the same accessibility level. If not, it returns False. 12.2/2

For the purposes of the dynamic semantics of functions Descendant_Tag and Is_Descendant_At_Same_Level, a tagged type T2 is a *descendant* of a type T1 if it is the same as T1, or if its parent type or one of its progenitor types is a descendant of type T1 by this rule, even if at the point of the declaration of T2, one of the derivations in the chain is not visible. 12.3/3

The function Parent_Tag returns the tag of the parent type of the type whose tag is T. If the type does not have a parent type (that is, it was not defined by a derived_type_definition), then No_Tag is returned. 12.4/4

12.5/3 The function Interface_Ancestor_Tags returns an array containing the tag of each interface ancestor type of the type whose tag is T, other than T itself. The lower bound of the returned array is 1, and the order of the returned tags is unspecified. Each tag appears in the result exactly once. If the type whose tag is T has no interface ancestors, a null array is returned.

12.6/3 The function Is_Abstract returns True if the type whose tag is T is abstract, and False otherwise.

13 For every subtype S of a tagged type *T* (specific or class-wide), the following attributes are defined:

14 S'Class S'Class denotes a subtype of the class-wide type (called *T*'Class in this document) for the class rooted at *T* (or if S already denotes a class-wide subtype, then S'Class is the same as S).

15 S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type *T* belong to S.

16 S'Tag S'Tag denotes the tag of the type *T* (or if *T* is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag.

17 Given a prefix X that is of a class-wide tagged type (after any implicit dereference), the following attribute is defined:

18 X'Tag X'Tag denotes the tag of X. The value of this attribute is of type Tag.

18.1/2 The following language-defined generic function exists:

18.2/5
```
generic
    type T (<>) is abstract tagged limited private;
    type Parameters (<>) is limited private;
    with function Constructor (Params : not null access Parameters)
        return T is abstract;
function Ada.Tags.Generic_Dispatching_Constructor
   (The_Tag : Tag;
    Params  : not null access Parameters) return T'Class
   with Preelaborate, Convention => Intrinsic,
        Nonblocking, Global => in out synchronized;
```

18.3/2 Tags.Generic_Dispatching_Constructor provides a mechanism to create an object of an appropriate type from just a tag value. The function Constructor is expected to create the object given a reference to an object of type Parameters.

Dynamic Semantics

19 The tag associated with an object of a tagged type is determined as follows:

20 - The tag of a stand-alone object, a component, or an aggregate of a specific tagged type *T* identifies *T*.

21/5 - The tag of an object created by an allocator for an access type with a specific designated tagged type *T* identifies *T*.

22 - The tag of an object of a class-wide tagged type is that of its initialization expression.

23 - The tag of the result returned by a function whose result type is a specific tagged type *T* identifies *T*.

24/2 - The tag of the result returned by a function with a class-wide result type is that of the return object.

25 The tag is preserved by type conversion and by parameter passing. The tag of a value is the tag of the associated object (see 6.2).

Tag_Error is raised by a call of Descendant_Tag, Expanded_Name, External_Tag, Interface_Ancestor_Tags, Is_Abstract, Is_Descendant_At_Same_Level, Parent_Tag, Wide_Expanded_Name, or Wide_Wide_Expanded_Name if any tag passed is No_Tag. 25.1/3

An instance of Tags.Generic_Dispatching_Constructor raises Tag_Error if The_Tag does not represent a concrete descendant of T or if the innermost master (see 7.6.1) of this descendant is not also a master of the instance. Otherwise, it dispatches to the primitive function denoted by the formal Constructor for the type identified by The_Tag, passing Params, and returns the result. Any exception raised by the function is propagated. 25.2/2

Erroneous Execution

If an internal tag provided to an instance of Tags.Generic_Dispatching_Constructor or to any subprogram declared in package Tags identifies either a type that is not library-level and whose tag has not been created (see 13.14), or a type that does not exist in the partition at the time of the call, then execution is erroneous. 25.3/2

Implementation Permissions

The implementation of Internal_Tag and Descendant_Tag may raise Tag_Error if no specific type corresponding to the string External passed as a parameter exists in the partition at the time the function is called, or if there is no such type whose innermost master is a master of the point of the function call. 26/2

Implementation Advice

Internal_Tag should return the tag of a type, if one exists, whose innermost master is a master of the point of evaluation of the function call. 26.1/6

NOTE 1 A type declared with the reserved word **tagged** is normally declared in a package_specification, so that new primitive subprograms can be declared for it. 27/5

NOTE 2 Once an object has been created, its tag never changes. 28

NOTE 3 Class-wide types are defined to have unknown discriminants (see 3.7). This means that, by the rules in 3.7 for objects with unknown discriminants, objects of a class-wide type are illegal unless they are explicitly initialized (whether created by an object_declaration or an allocator), and that aggregates are illegal unless they are explicitly qualified with a specific type when their expected type is class-wide. 29/5

NOTE 4 The capability provided by Tags.Generic_Dispatching_Constructor is sometimes known as a *factory*. 30/2

Examples

Examples of tagged record types: 31

```
type Point is tagged                                                    32
  record
    X, Y : Real := 0.0;
  end record;

type Expression is tagged null record;                                  33
  -- Components will be added by each extension
```

3.9.1 Type Extensions

Every type extension is a tagged type, and is a *record extension* or a *private extension* of some other tagged type, or a noninterface synchronized tagged type. 1/2

Syntax

record_extension_part ::= **with** record_definition 2

Legality Rules

3/2 The parent type of a record extension shall not be a class-wide type nor shall it be a synchronized tagged type (see 3.9.4). If the parent type or any progenitor is nonlimited, then each of the components of the record_extension_part shall be nonlimited. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

4/2 Within the body of a generic unit, or the body of any of its descendant library units, a tagged type shall not be declared as a descendant of a formal type declared within the formal part of the generic unit.

Static Semantics

4.1/2 A record extension is a *null extension* if its declaration has no known_discriminant_part and its record_extension_part includes no component_declarations.

4.2/5 In the case where the (compile-time) view of an object *X* is of a tagged type T1 or T1'Class and the (run-time) tag of *X* is T2'Tag, only the components (if any) of *X* that are components of T1 (or that are discriminants which correspond to a discriminant of T1) are said to be *components of the nominal type* of *X*. Similarly, only parts (respectively, subcomponents) of T1 are parts (respectively, subcomponents) of the nominal type of *X*.

4.3/6 For a record extension or private extension that does not have a known_discriminant_part, the following type-related aspect can be specified:

4.4/6 Constructor_Extension

A type extension for which aspect Constructor_Extension is True is said to have *automatic constructor extension*. As defined in 3.4, for such a type, an inherited constructor function whose corresponding constructor function of the parent of the full type is not abstract, is defined to return an object given by an extension aggregate.

4.5/6 Aspect Constructor_Extension is never inherited; it is True if the type extension is a null extension or if the aspect is specified as True for the type extension; it is False otherwise. If a type has a partial view that is a private extension, the Constructor_Extension aspect shall be specified (if at all) only on the partial view. If specified True on the partial view, the full type shall not have a known_discriminant_part.

Dynamic Semantics

5 The elaboration of a record_extension_part consists of the elaboration of the record_definition.

6 NOTE 1 The term “type extension” refers to a type as a whole. The term “extension part” refers to the piece of text that defines the additional components (if any) the type extension has relative to its specified ancestor type.

7/2 NOTE 2 When an extension is declared immediately within a body, primitive subprograms are inherited and are overridable, but new primitive subprograms cannot be added.

8/5 NOTE 3 By the rules given in 3.8, a name that denotes a component (including a discriminant) of the parent type is illegal within the record_extension_part. Similarly, a name that denotes a component defined within the record_extension_part is illegal within the record_extension_part. A name that denotes a discriminant of the record extension is legal, providing that it refers to a discriminant defined in a new known_discriminant_part in the enclosing type declaration.

9/5 NOTE 4 By the rules given in 8.3, each visible component of a record extension will have a unique name, whether the component is (visibly) inherited from the parent type or declared in the record_extension_part.

Examples

Examples of record extensions (of types defined above in 3.9):

```
type Painted_Point is new Point with
  record
    Paint : Color := White;
  end record;
    -- Components X and Y are inherited

Origin : constant Painted_Point := (X | Y => 0.0, Paint => Black);

type Literal is new Expression with
  record                -- a leaf in an Expression tree
    Value : Real;
  end record;

type Expr_Ptr is access all Expression'Class;
                                -- see 3.9

type Binary_Operation is new Expression with
  record                -- an internal node in an Expression tree
    Left, Right : Expr_Ptr;
  end record;

type Addition is new Binary_Operation with null record;
type Subtraction is new Binary_Operation with null record;
  -- No additional components needed for these extensions

Tree : Expr_Ptr :=      -- A tree representation of "5.0 + (13.0–7.0)"
   new Addition'(
      Left  => new Literal'(Value => 5.0),
      Right => new Subtraction'(
         Left  => new Literal'(Value => 13.0),
         Right => new Literal'(Value => 7.0)));
```

3.9.2 Dispatching Operations of Tagged Types

The primitive subprograms of a tagged type, the subprograms declared by formal_abstract_subprogram_declarations, the Put_Image attribute (see 4.10) of a specific tagged type, and the stream attributes of a specific tagged type that are available (see 13.13.2) at the end of the declaration list where the type is declared are called *dispatching operations*. A dispatching operation can be called using a statically determined *controlling* tag, in which case the body to be executed is determined at compile time. Alternatively, the controlling tag can be dynamically determined, in which case the call *dispatches* to a body that is determined at run time; such a call is termed a *dispatching call*. As explained below, the properties of the operands and the context of a particular call on a dispatching operation determine how the controlling tag is determined, and hence whether or not the call is a dispatching call. Run-time polymorphism is achieved when a dispatching operation is called by a dispatching call.

Static Semantics

A *call on a dispatching operation* is a call whose name or prefix denotes the declaration of a dispatching operation. A *controlling operand* in a call on a dispatching operation of a tagged type *T* is one whose corresponding formal parameter is of type *T* or is of an anonymous access type with designated type *T*; the corresponding formal parameter is called a *controlling formal parameter*. If the controlling formal parameter is an access parameter, the controlling operand is the object designated by the actual parameter, rather than the actual parameter itself. If the call is to a (primitive) function with result type *T* (a *function with a controlling result*), then the call has a *controlling result* — the context of the call can control the dispatching. Similarly, if the call is to a function with an access result type designating *T* (a *function with a controlling access result*), then the call has a *controlling access result*, and the context can similarly control dispatching.

3/5 A name or expression of a tagged type is either *statically* tagged, *dynamically* tagged, or *tag indeterminate*, according to whether, when used as a controlling operand, the tag that controls dispatching is determined statically by the operand's (specific) type, dynamically by its tag at run time, or from context. A qualified_expression or parenthesized expression is statically, dynamically, or indeterminately tagged according to its operand. A conditional_expression is statically, dynamically, or indeterminately tagged according to rules given in 4.5.7. A declare_expression is statically, dynamically, or indeterminately tagged according to its *body*_expression. For other kinds of names and expressions, this is determined as follows:

4/2 - The name or expression is *statically tagged* if it is of a specific tagged type and, if it is a call with a controlling result or controlling access result, it has at least one statically tagged controlling operand;

5/2 - The name or expression is *dynamically tagged* if it is of a class-wide type, or it is a call with a controlling result or controlling access result and at least one dynamically tagged controlling operand;

6/2 - The name or expression is *tag indeterminate* if it is a call with a controlling result or controlling access result, all of whose controlling operands (if any) are tag indeterminate.

7/1 A type_conversion is statically or dynamically tagged according to whether the type determined by the subtype_mark is specific or class-wide, respectively. For an object that is designated by an expression whose expected type is an anonymous access-to-specific tagged type, the object is dynamically tagged if the expression, ignoring enclosing parentheses, is of the form X'Access, where X is of a class-wide type, or is of the form **new** T'(...), where T denotes a class-wide subtype. Otherwise, the object is statically or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively.

Legality Rules

8 A call on a dispatching operation shall not have both dynamically tagged and statically tagged controlling operands.

9/1 If the expected type for an expression or name is some specific tagged type, then the expression or name shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation. Similarly, if the expected type for an expression is an anonymous access-to-specific tagged type, then the object designated by the expression shall not be dynamically tagged unless it is a controlling operand in a call on a dispatching operation.

10/2 In the declaration of a dispatching operation of a tagged type, everywhere a subtype of the tagged type appears as a subtype of the profile (see 6.1), it shall statically match the first subtype of the tagged type. If the dispatching operation overrides an inherited subprogram, it shall be subtype conformant with the inherited subprogram. The convention of an inherited dispatching operation is the convention of the corresponding primitive operation of the parent or progenitor type. The default convention of a dispatching operation that overrides an inherited primitive operation is the convention of the inherited operation; if the operation overrides multiple inherited operations, then they shall all have the same convention. An explicitly declared dispatching operation shall not be of convention Intrinsic.

11/2 The default_expression for a controlling formal parameter of a dispatching operation shall be tag indeterminate.

11.1/2 If a dispatching operation is defined by a subprogram_renaming_declaration or the instantiation of a generic subprogram, any access parameter of the renamed subprogram or the generic subprogram that

corresponds to a controlling access parameter of the dispatching operation, shall have a subtype that excludes null.

A given subprogram shall not be a dispatching operation of two or more distinct tagged types. 12

The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see 13.14). For example, new dispatching operations cannot be added after objects or values of the type exist, nor after deriving a record extension from it, nor after a body. 13

Dynamic Semantics

For the execution of a call on a dispatching operation of a type *T*, the *controlling tag value* determines which subprogram body is executed. The controlling tag value is defined as follows: 14

- If one or more controlling operands are statically tagged, then the controlling tag value is *statically determined* to be the tag of *T*. 15
- If one or more controlling operands are dynamically tagged, then the controlling tag value is not statically determined, but is rather determined by the tags of the controlling operands. If there is more than one dynamically tagged controlling operand, a check is made that they all have the same tag. If this check fails, Constraint_Error is raised unless the call is a function_call whose name denotes the declaration of an equality operator (predefined or user defined) that returns Boolean, in which case the result of the call is defined to indicate inequality, and no subprogram_body is executed. This check is performed prior to evaluating any tag-indeterminate controlling operands. 16
- If all of the controlling operands (if any) are tag-indeterminate, then: 17/2
  - If the call has a controlling result or controlling access result that is itself, or designates, the evaluated operative constituent (see 4.4) of: 18/6
    - a tag-indeterminate controlling operand of an enclosing call on a dispatching operation of a descendant of type *T*, then the controlling tag value is determined by the controlling tag value of this enclosing call; 18.1/6
    - a tag-indeterminate expression of an assignment_statement whose target is of a class-wide type, then the controlling tag value is determined by the target; 18.2/6
  - 18.3/6
  - Otherwise, the controlling tag value is statically determined to be the tag of type *T*. 19

For the execution of a call on a dispatching operation, the action performed is determined by the properties of the corresponding dispatching operation of the specific type identified by the controlling tag value: 20/3

- if the corresponding operation is explicitly declared for this type, even if the declaration occurs in a private part, then the action comprises an invocation of the explicit body for the operation; 20.1/3
- if the corresponding operation is implicitly declared for this type and is implemented by an entry or protected subprogram (see 9.1 and 9.4), then the action comprises a call on this entry or protected subprogram, with the target object being given by the first actual parameter of the call, and the actual parameters of the entry or protected subprogram being given by the remaining actual parameters of the call, if any; 20.2/3
- if the corresponding operation is a predefined operator then the action comprises an invocation of that operator; 20.3/3
- otherwise, the action is the same as the action for the corresponding operation of the parent type or progenitor type from which the operation was inherited except that additional invariant checks (see 7.3.2) and class-wide postcondition checks (see 6.1.1) may apply. If there is more than one 20.4/3

such corresponding operation, the action is that for the operation that is not a null procedure, if any; otherwise, the action is that of an arbitrary one of the operations.

21 NOTE 1 The body to be executed for a call on a dispatching operation is determined by the tag; it does not matter whether that tag is determined statically or dynamically, and it does not matter whether the subprogram's declaration is visible at the place of the call.

22/2 NOTE 2 This subclause covers calls on dispatching subprograms of a tagged type. Rules for tagged type membership tests are described in 4.5.2. Controlling tag determination for an assignment_statement is described in 5.2.

23 NOTE 3 A dispatching call can dispatch to a body whose declaration is not visible at the place of the call.

24 NOTE 4 A call through an access-to-subprogram value is never a dispatching call, even if the access value designates a dispatching operation. Similarly a call whose prefix denotes a subprogram_renaming_declaration cannot be a dispatching call unless the renaming itself is the declaration of a primitive subprogram.

3.9.3 Abstract Types and Subprograms

1/2 An *abstract type* is a tagged type intended for use as an ancestor of other types, but which is not allowed to have objects of its own. An *abstract subprogram* is a subprogram that has no body, but is intended to be overridden at some point when inherited. Because objects of an abstract type cannot be created, a dispatching call to an abstract subprogram always dispatches to some overriding body.

Syntax

1.1/3
```
abstract_subprogram_declaration ::=
   [overriding_indicator]
   subprogram_specification is abstract
      [aspect_specification];
```

Static Semantics

1.2/2 Interface types (see 3.9.4) are abstract types. In addition, a tagged type that has the reserved word **abstract** in its declaration is an abstract type. The class-wide type (see 3.4.1) rooted at an abstract type is not itself an abstract type.

Legality Rules

2/2 Only a tagged type shall have the reserved word **abstract** in its declaration.

3/2 A subprogram declared by an abstract_subprogram_declaration or a formal_abstract_subprogram_declaration (see 12.6) is an *abstract subprogram*. If it is a primitive subprogram of a tagged type, then the tagged type shall be abstract.

4/6 If a type has an implicitly declared primitive subprogram that is inherited or is a predefined operator, and the corresponding primitive subprogram of the parent or ancestor type is abstract or is a function with a controlling access result, or if a nonabstract type that does not have automatic constructor extension inherits a constructor function, then:

5/2 - If the type is abstract or untagged, the implicitly declared subprogram is *abstract*.

6/6 - Otherwise, the subprogram shall be overridden with a nonabstract subprogram or, in the case of a private extension inheriting a nonabstract constructor function, have a full type that has automatic constructor extension; for a type declared in the visible part of a package, the overriding may be either in the visible or the private part. Such a subprogram is said to *require overriding*. However, if the type is a generic formal type, the subprogram is allowed to be inherited as is, without being overridden for the formal type itself; a nonabstract version will necessarily be provided by the actual type.

A call on an abstract subprogram shall be a dispatching call; nondispatching calls to an abstract subprogram are not allowed. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 7/5

If the name or prefix given in an iterator_procedure_call (see 5.5.3) denotes an abstract subprogram, the subprogram shall be a dispatching subprogram. 7.1/5

The type of an aggregate, of an object created by an object_declaration or an allocator, or of a generic formal object of mode **in**, shall not be abstract. The type of the target of an assignment operation (see 5.2) shall not be abstract. The type of a component shall not be abstract. If the result type of a function is abstract, then the function shall be abstract. If a function has an access result type designating an abstract type, then the function shall be abstract. The type denoted by a return_subtype_indication (see 6.5) shall not be abstract. A generic function shall not have an abstract result type or an access result type designating an abstract type. 8/5

If a partial view is not abstract, the corresponding full view shall not be abstract. If a generic formal type is abstract, then for each primitive subprogram of the formal that is not abstract, the corresponding primitive subprogram of the actual shall not be abstract. 9

For an abstract type declared in a visible part, an abstract primitive subprogram shall not be declared in the private part, unless it is overriding an abstract subprogram implicitly declared in the visible part. For a tagged type declared in a visible part, a primitive function with a controlling result or a controlling access result shall not be declared in the private part, unless it is overriding a function implicitly declared in the visible part. 10/3

A generic actual subprogram shall not be an abstract subprogram unless the generic formal subprogram is declared by a formal_abstract_subprogram_declaration. The prefix of an attribute_reference for the Access, Unchecked_Access, or Address attributes shall not denote an abstract subprogram. 11/2

Dynamic Semantics

The elaboration of an abstract_subprogram_declaration has no effect. 11.1/2

NOTE 1 Abstractness is not inherited; a type is abstract only if the reserved word **abstract** is used in the declaration of the type extension. 12/5

NOTE 2 A class-wide type is never abstract. Even if a class is rooted at an abstract type, the class-wide type for the class is not abstract, and an object of the class-wide type can be created; the tag of such an object will identify some nonabstract type in the class. 13

Examples

Example of an abstract type representing a set of natural numbers: 14

```
package Sets is                                                                    15
    subtype Element_Type is Natural;
    type Set is abstract tagged null record;
    function Empty return Set is abstract;
    function Union(Left, Right : Set) return Set is abstract;
    function Intersection(Left, Right : Set) return Set is abstract;
    function Unit_Set(Element : Element_Type) return Set is abstract;
    procedure Take(Element : out Element_Type;
                   From : in out Set) is abstract;
end Sets;
```

Given the above abstract type, one can derive various (nonabstract) extensions of the type, representing alternative implementations of a set. One possibility is to use a bit vector, but impose an upper bound on the largest element representable, while another possible implementation is a hash table, trading off space for flexibility. 16/5

3.9.4 Interface Types

1/2 An interface type is an abstract tagged type that provides a restricted form of multiple inheritance. A tagged type, task type, or protected type may have one or more interface types as ancestors.

Syntax

2/2 interface_type_definition ::=
[**limited** | **task** | **protected** | **synchronized**] **interface** [**and** interface_list]

3/2 interface_list ::= *interface*_subtype_mark {**and** *interface*_subtype_mark}

Static Semantics

4/2 An interface type (also called an *interface*) is a specific abstract tagged type that is defined by an interface_type_definition.

5/2 An interface with the reserved word **limited**, **task**, **protected**, or **synchronized** in its definition is termed, respectively, a *limited interface*, a *task interface*, a *protected interface*, or a *synchronized interface*. In addition, all task and protected interfaces are synchronized interfaces, and all synchronized interfaces are limited interfaces.

6/2 A task or protected type derived from an interface is a tagged type. Such a tagged type is called a *synchronized* tagged type, as are synchronized interfaces and private extensions whose declaration includes the reserved word **synchronized**.

7/2 A task interface is an abstract task type. A protected interface is an abstract protected type.

8/2 An interface type has no components.

9/2 An *interface*_subtype_mark in an interface_list names a *progenitor subtype*; its type is the *progenitor type*. An interface type inherits user-defined primitive subprograms from each progenitor type in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4).

Legality Rules

10/2 All user-defined primitive subprograms of an interface type shall be abstract subprograms or null procedures.

11/2 The type of a subtype named in an interface_list shall be an interface type.

12/2 A type derived from a nonlimited interface shall be nonlimited.

13/2 An interface derived from a task interface shall include the reserved word **task** in its definition; any other type derived from a task interface shall be a private extension or a task type declared by a task declaration (see 9.1).

14/2 An interface derived from a protected interface shall include the reserved word **protected** in its definition; any other type derived from a protected interface shall be a private extension or a protected type declared by a protected declaration (see 9.4).

15/2 An interface derived from a synchronized interface shall include one of the reserved words **task**, **protected**, or **synchronized** in its definition; any other type derived from a synchronized interface shall be a private extension, a task type declared by a task declaration, or a protected type declared by a protected declaration.

No type shall be derived from both a task interface and a protected interface. 16/2

In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 17/2

Dynamic Semantics

The elaboration of an interface_type_definition creates the interface type and its first subtype. 18/3

NOTE Nonlimited interface types have predefined nonabstract equality operators. These can be overridden with user-defined abstract equality operators. Such operators will then require an explicit overriding for any nonabstract descendant of the interface. 19/5

Examples

Example of a limited interface and a synchronized interface extending it: 20/2

```
type Queue is limited interface;                                                    21/2
procedure Append(Q : in out Queue; Person : in Person_Name) is abstract;
procedure Remove_First(Q      : in out Queue;
                       Person :    out Person_Name) is abstract;
function Cur_Count(Q : in Queue) return Natural is abstract;
function Max_Count(Q : in Queue) return Natural is abstract;
-- See 3.10.1 for Person_Name.

Queue_Error : exception;                                                            22/3
-- Append raises Queue_Error if Cur_Count(Q) = Max_Count(Q)
-- Remove_First raises Queue_Error if Cur_Count(Q) = 0

type Synchronized_Queue is                                                          23/2
                       synchronized interface and Queue; -- see 9.11
procedure Append_Wait(Q      : in out Synchronized_Queue;
                      Person : in     Person_Name) is abstract;
procedure Remove_First_Wait(Q      : in out Synchronized_Queue;
                            Person :    out Person_Name) is abstract;

...                                                                                 24/2

procedure Transfer(From   : in out Queue'Class;                                     25/2
                   To     : in out Queue'Class;
                   Number : in     Natural := 1) is
   Person : Person_Name;
begin
   for I in 1..Number loop
      Remove_First(From, Person);
      Append(To, Person);
   end loop;
end Transfer;
```

This defines a Queue interface defining a queue of people. (A similar design is possible to define any kind of queue simply by replacing Person_Name by an appropriate type.) The Queue interface has four dispatching operations, Append, Remove_First, Cur_Count, and Max_Count. The body of a class-wide operation, Transfer is also shown. Every nonabstract extension of Queue will provide implementations for at least its four dispatching operations, as they are abstract. Any object of a type derived from Queue can be passed to Transfer as either the From or the To operand. The two operands can be of different types in a given call. 26/5

The Synchronized_Queue interface inherits the four dispatching operations from Queue and adds two additional dispatching operations, which wait if necessary rather than raising the Queue_Error exception. This synchronized interface can only be implemented by a task or protected type, and as such ensures safe concurrent access. 27/5

Example use of the interface: 28/2

29/3
```
type Fast_Food_Queue is new Queue with record ...;
procedure Append(Q : in out Fast_Food_Queue; Person : in Person_Name);
procedure Remove_First(Q : in out Fast_Food_Queue;
                       Person : out Person_Name);
function Cur_Count(Q : in Fast_Food_Queue) return Natural;
function Max_Count(Q : in Fast_Food_Queue) return Natural;
```

30/2
```
...
```

31/2
```
Cashier, Counter : Fast_Food_Queue;
```

32/5
```
...
-- Add Casey (see 3.10.1) to the cashier's queue:
Append (Cashier, Casey);
-- After payment, move Casey to the sandwich counter queue:
Transfer (Cashier, Counter);
...
```

33/5 An interface such as Queue can be used directly as the parent of a new type (as shown here), or can be used as a progenitor when a type is derived. In either case, the primitive operations of the interface are inherited. For Queue, the implementation of the four inherited routines will necessarily be provided. Inside the call of Transfer, calls will dispatch to the implementations of Append and Remove_First for type Fast_Food_Queue.

34/2 *Example of a task interface:*

35/2
```
type Serial_Device is task interface;  -- see 9.1
procedure Read (Dev : in Serial_Device; C : out Character) is abstract;
procedure Write(Dev : in Serial_Device; C : in  Character) is abstract;
```

36/2 The Serial_Device interface has two dispatching operations which are intended to be implemented by task entries (see 9.1).

3.10 Access Types

1 A value of an access type (an *access value*) provides indirect access to the object or subprogram it *designates*. Depending on its type, an access value can designate either subprograms, objects created by allocators (see 4.8), or more generally *aliased* objects of an appropriate type.

Syntax

2/2 access_type_definition ::=
[null_exclusion] access_to_object_definition
| [null_exclusion] access_to_subprogram_definition

3 access_to_object_definition ::=
access [general_access_modifier] subtype_indication

4 general_access_modifier ::= **all** | **constant**

5 access_to_subprogram_definition ::=
access [**protected**] **procedure** parameter_profile
| **access** [**protected**] **function** parameter_and_result_profile

5.1/2 null_exclusion ::= **not null**

6/2 access_definition ::=
[null_exclusion] **access** [**constant**] subtype_mark
| [null_exclusion] **access** [**protected**] **procedure** parameter_profile
| [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile

Static Semantics

There are two kinds of access types, *access-to-object* types, whose values designate objects, and *access-to-subprogram* types, whose values designate subprograms. Associated with an access-to-object type is a *storage pool*; several access types may share the same storage pool. All descendants of an access type share the same storage pool. A storage pool is an area of storage used to hold dynamically allocated objects (called *pool elements*) created by allocators; storage pools are described further in 13.11, "Storage Management". 7/1

Access-to-object types are further subdivided into *pool-specific* access types, whose values can designate only the elements of their associated storage pool, and *general* access types, whose values can designate the elements of any storage pool, as well as aliased objects created by declarations rather than allocators, and aliased subcomponents of other objects. 8

A view of an object is defined to be *aliased* if it is defined by an object_declaration, component_definition, parameter_specification, or extended_return_object_declaration with the reserved word **aliased**, or by a renaming of an aliased view. In addition, the dereference of an access-to-object value denotes an aliased view, as does a view conversion (see 4.6) of an aliased view. A qualified_expression denotes an aliased view when the operand denotes an aliased view. The current instance of an immutably limited type (see 7.5) is defined to be aliased. Finally, a formal parameter or generic formal object of a tagged type is defined to be aliased. Aliased views are the ones that can be designated by an access value. 9/5

An access_to_object_definition defines an access-to-object type and its first subtype; the subtype_indication defines the *designated subtype* of the access type. If a general_access_modifier appears, then the access type is a general access type. If the modifier is the reserved word **constant**, then the type is an *access-to-constant type*; a designated object cannot be updated through a value of such a type. If the modifier is the reserved word **all**, then the type is an *access-to-variable type*; a designated object can be both read and updated through a value of such a type. If no general_access_modifier appears in the access_to_object_definition, the access type is a pool-specific access-to-variable type. 10

An access_to_subprogram_definition defines an access-to-subprogram type and its first subtype; the parameter_profile or parameter_and_result_profile defines the *designated profile* of the access type. There is a *calling convention* associated with the designated profile; only subprograms with this calling convention can be designated by values of the access type. By default, the calling convention is "*protected*" if the reserved word **protected** appears, and "Ada" otherwise. See Annex B for how to override this default. 11

An access_definition defines an anonymous general access type or an anonymous access-to-subprogram type. For a general access type, the subtype_mark denotes its *designated subtype*; if the general_access_modifier **constant** appears, the type is an access-to-constant type; otherwise, it is an access-to-variable type. For an access-to-subprogram type, the parameter_profile or parameter_and_result_profile denotes its *designated profile*. 12/3

For each access type, there is a null access value designating no entity at all, which can be obtained by (implicitly) converting the literal **null** to the access type. The null value of an access type is the default initial value of the type. Nonnull values of an access-to-object type are obtained by evaluating an allocator, which returns an access value designating a newly created object (see 3.10.2), or in the case of a general access-to-object type, evaluating an attribute_reference for the Access or Unchecked_Access attribute of an aliased view of an object. Nonnull values of an access-to-subprogram type are obtained by evaluating an attribute_reference for the Access attribute of a nonintrinsic subprogram. 13/2

A null_exclusion in a construct specifies that the null value does not belong to the access subtype defined by the construct, that is, the access subtype *excludes null*. In addition, the anonymous access subtype 13.1/2

defined by the access_definition for a controlling access parameter (see 3.9.2) excludes null. Finally, for a subtype_indication without a null_exclusion, the subtype denoted by the subtype_indication excludes null if and only if the subtype denoted by the subtype_mark in the subtype_indication excludes null.

14/3 All subtypes of an access-to-subprogram type are constrained. The first subtype of a type defined by an access_definition or an access_to_object_definition is unconstrained if the designated subtype is an unconstrained array or discriminated subtype; otherwise, it is constrained.

Legality Rules

14.1/2 If a subtype_indication, discriminant_specification, parameter_specification, parameter_and_result_profile, object_renaming_declaration, or formal_object_declaration has a null_exclusion, the subtype_mark in that construct shall denote an access subtype that does not exclude null.

Dynamic Semantics

15/2 A composite_constraint is *compatible* with an unconstrained access subtype if it is compatible with the designated subtype. A null_exclusion is compatible with any access subtype that does not exclude null. An access value *satisfies* a composite_constraint of an access subtype if it equals the null value of its type or if it designates an object whose value satisfies the constraint. An access value satisfies an exclusion of the null value if it does not equal the null value of its type.

16 The elaboration of an access_type_definition creates the access type and its first subtype. For an access-to-object type, this elaboration includes the elaboration of the subtype_indication, which creates the designated subtype.

17/2 The elaboration of an access_definition creates an anonymous access type.

18 NOTE 1 Access values are called “pointers” or “references” in some other languages.

19 NOTE 2 Each access-to-object type has an associated storage pool; several access types can share the same pool. An object can be created in the storage pool of an access type by an allocator (see 4.8) for the access type. A storage pool (roughly) corresponds to what some other languages call a “heap”. See 13.11 for a discussion of pools.

20 NOTE 3 Only index_constraints and discriminant_constraints can be applied to access types (see 3.6.1 and 3.7.1).

Examples

21 *Examples of access-to-object types:*

22/4
```
type Frame is access Matrix;  -- see 3.6
type Peripheral_Ref is not null access Peripheral;  -- see 3.8.1
type Binop_Ptr is access all Binary_Operation'Class;
                                -- general access-to-class-wide, see 3.9.1
```

23 *Example of an access subtype:*

24
```
subtype Drum_Ref is Peripheral_Ref(Drum); -- see 3.8.1
```

25 *Example of an access-to-subprogram type:*

26
```
type Message_Procedure is access procedure (M : in String := "Error!");
procedure Default_Message_Procedure(M : in String);
Give_Message : Message_Procedure := Default_Message_Procedure'Access;
...
procedure Other_Procedure(M : in String);
...
Give_Message := Other_Procedure'Access;
...
Give_Message("File not found.");
                          -- call with parameter (.all is optional)
Give_Message.all;         -- call with no parameters
```

3.10.1 Incomplete Type Declarations

There are no particular limitations on the designated type of an access type. In particular, the type of a component of the designated type can be another access type, or even the same access type. This permits mutually dependent and recursive access types. An incomplete_type_declaration can be used to introduce a type to be used as a designated type, while deferring its full definition to a subsequent full_type_declaration. 1

Syntax

incomplete_type_declaration ::= 2/2
  type defining_identifier [discriminant_part] [**is tagged**];

Static Semantics

An incomplete_type_declaration declares an *incomplete view* of a type and its first subtype; the first subtype is unconstrained if a discriminant_part appears. If the incomplete_type_declaration includes the reserved word **tagged**, it declares a *tagged incomplete view*. If *T* denotes a tagged incomplete view, then *T*'Class denotes a tagged incomplete view. An incomplete view of a type is a limited view of the type (see 7.5). 2.1/4

Given an access type *A* whose designated type *T* is an incomplete view, a dereference of a value of type *A* also has this incomplete view except when: 2.2/2

- it occurs within the immediate scope of the completion of *T*, or 2.3/2
- it occurs within the scope of a nonlimited_with_clause that mentions a library package in whose visible part the completion of *T* is declared, or 2.4/3
- it occurs within the scope of the completion of *T* and *T* is an incomplete view declared by an incomplete_type_declaration. 2.5/3

In these cases, the dereference has the view of *T* visible at the point of the dereference. 2.6/3

Similarly, if a subtype_mark denotes a subtype_declaration defining a subtype of an incomplete view *T*, the subtype_mark denotes an incomplete view except under the same three circumstances given above, in which case it denotes the view of *T* visible at the point of the subtype_mark. 2.7/3

Legality Rules

An incomplete_type_declaration requires a completion, which shall be a type_declaration other than an incomplete_type_declaration. If the incomplete_type_declaration occurs immediately within either the visible part of a package_specification or a declarative_part, then the type_declaration shall occur later and immediately within this visible part or declarative_part. If the incomplete_type_declaration occurs immediately within the private part of a given package_specification, then the type_declaration shall occur later and immediately within either the private part itself, or the declarative_part of the corresponding package_body. 3/3

If an incomplete_type_declaration includes the reserved word **tagged**, then a type_declaration that completes it shall declare a tagged type. If an incomplete_type_declaration has a known_discriminant_part, then a type_declaration that completes it shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). If an incomplete_type_declaration has no discriminant_part (or an unknown_discriminant_part), then a corresponding type_declaration is nevertheless allowed to have discriminants, either explicitly, or inherited via derivation. 4/3

5/2 A name that denotes an incomplete view of a type may be used as follows:

6/3 • as the subtype_mark in the subtype_indication of an access_to_object_definition; the only form of constraint allowed in this subtype_indication is a discriminant_constraint (a null_exclusion is not allowed);

7/2 • as the subtype_mark in the subtype_indication of a subtype_declaration; the subtype_indication shall not have a null_exclusion or a constraint;

8/3 • as the subtype_mark in an access_definition for an access-to-object type;

8.1/6 • as the subtype_mark defining the subtype of a parameter or result in a profile occurring within a basic_declaration other than a renaming-as-body (see 8.5.4);

8.2/3 • as a generic actual parameter whose corresponding generic formal parameter is a formal incomplete type (see 12.5.1).

8.3/2 If such a name denotes a tagged incomplete view, it may also be used:

8.4/6 • as the subtype_mark defining the subtype of a parameter in the profile for a subprogram_body, entry_body, accept_statement, or renaming-as-body;

9/2 • as the prefix of an attribute_reference whose attribute_designator is Class; such an attribute_reference is restricted to the uses allowed here; it denotes a tagged incomplete view.

9.1/3 *This paragraph was deleted.*

9.2/3 • *This paragraph was deleted.*

9.3/2 If any of the above uses occurs as part of the declaration of a primitive subprogram of the incomplete view, and the declaration occurs immediately within the private part of a package, then the completion of the incomplete view shall also occur immediately within the private part; it shall not be deferred to the package body.

9.4/2 No other uses of a name that denotes an incomplete view of a type are allowed.

10/3 A prefix that denotes an object shall not be of an incomplete view. An actual parameter in a call shall not be of an untagged incomplete view. The result object of a function call shall not be of an incomplete view. A prefix shall not denote a subprogram having a formal parameter of an untagged incomplete view, nor a return type that is an incomplete view.

10.1/5 The controlling operand or controlling result of a dispatching call shall not be of an incomplete view if the operand or result is dynamically tagged.

Paragraph 11 was deleted.

Dynamic Semantics

12 The elaboration of an incomplete_type_declaration has no effect.

13/5 NOTE 1 Within a declarative_part, an incomplete_type_declaration and a corresponding full_type_declaration cannot be separated by an intervening body. This is because, by the rules given in 13.14, a type is illegal if it is not completely defined before it is frozen, and a body freezes all types declared prior to it in the same declarative_part.

13.1/3 NOTE 2 A name that denotes an object of an incomplete view is defined to be of a limited type. Hence, the target of an assignment statement cannot be of an incomplete view.

Examples

14 *Example of a recursive type:*

15
```
type Cell;  -- incomplete type declaration
type Link is access Cell;
```

```
type Cell is                                                                    16
   record
      Value  : Integer;
      Succ   : Link;
      Pred   : Link;
   end record;
Head   : Link  := new Cell'(0, null, null);                                     17
Next   : Link  := Head.Succ;
```

Examples of mutually dependent access types: 18

```
type Person(<>);     -- incomplete type declaration                             19/2
type Car is tagged;  -- incomplete type declaration
type Person_Name is access Person;                                              20/2
type Car_Name    is access all Car'Class;
type Car is tagged                                                              21/2
   record
      Number  : Integer;
      Owner   : Person_Name;
   end record;
type Person(Sex : Gender) is                                                    22
   record
      Name     : String(1 .. 20);
      Birth    : Date;
      Age      : Integer range 0 .. 130;
      Vehicle  : Car_Name;
      case Sex is
         when M => Wife           : Person_Name(Sex => F);
         when F => Husband        : Person_Name(Sex => M);
      end case;
   end record;
My_Car, Your_Car, Next_Car : Car_Name := new Car;  -- see 4.8                   23/5
Casey : Person_Name := new Person(M);
   ...
Casey.Vehicle := Your_Car;
```

3.10.2 Operations of Access Types

The attribute Access is used to create access values designating aliased objects and nonintrinsic subprograms. The "accessibility" rules prevent dangling references (in the absence of uses of certain unchecked features — see Clause 13). 1

Name Resolution Rules

For an attribute_reference with attribute_designator Access (or Unchecked_Access — see 13.10), the expected type shall be a single access type *A* such that: 2/2

- *A* is an access-to-object type with designated type *D* and the type of the prefix is *D*'Class or is covered by *D*, or 2.1/2
- *A* is an access-to-subprogram type whose designated profile is type conformant with that of the prefix. 2.2/2

The prefix of such an attribute_reference is never interpreted as an implicit_dereference or a parameterless function_call (see 4.1.4). The designated type or profile of the expected type of the attribute_reference is the expected type or profile for the prefix. 2.3/2

Static Semantics

The accessibility rules, which prevent dangling references, are written in terms of *accessibility levels*, which reflect the run-time nesting of *masters*. As explained in 7.6.1, a master is the execution of a certain 3/5

construct (called a *master construct*), such as a subprogram_body. An accessibility level is *deeper than* another if it is more deeply nested at run time. For example, an object declared local to a called subprogram has a deeper accessibility level than an object declared local to the calling subprogram. The accessibility rules for access types require that the accessibility level of an object designated by an access value be no deeper than that of the access type. This ensures that the object will live at least as long as the access type, which in turn ensures that the access value cannot later designate an object that no longer exists. The Unchecked_Access attribute may be used to circumvent the accessibility rules.

4 A given accessibility level is said to be *statically deeper* than another if the given level is known at compile time (as defined below) to be deeper than the other for all possible executions. In most cases, accessibility is enforced at compile time by Legality Rules. Run-time accessibility checks are also used, since the Legality Rules do not cover certain cases involving access parameters and generic packages.

5/5 Each master, and each entity and view created by it, has an accessibility level; when two levels are defined to be the same, the accessibility levels of the two associated entities are said to be *tied* to each other. Accessibility levels are defined as follows:

6 • The accessibility level of a given master is deeper than that of each dynamically enclosing master, and deeper than that of each master upon which the task executing the given master directly depends (see 9.3).

7/5 • An entity or view defined by a declaration and created as part of its elaboration has the same accessibility level as the innermost master of the declaration except in the cases of renaming and derived access types described below. A formal parameter of a callable entity has the same accessibility level as the master representing the invocation of the entity.

8/5 • The accessibility level of a view of an object or subprogram defined by a renaming_declaration is the same as that of the renamed view, unless the renaming is of a formal subprogram, in which case the accessibility level is that of the instance.

9/2 • The accessibility level of a view conversion, qualified_expression, or parenthesized expression, is the same as that of the operand.

9.1/5 • The accessibility level of a conditional_expression (see 4.5.7) is the accessibility level of the evaluated *dependent_*expression.

9.2/5 • The accessibility level of a declare_expression (see 4.5.9) is the accessibility level of the *body_*expression.

10/4 • The accessibility level of an aggregate that is used (in its entirety) to directly initialize part of an object is that of the object being initialized. In other contexts, the accessibility level of an aggregate is that of the innermost master that evaluates the aggregate. Corresponding rules apply to a value conversion (see 4.6).

10.1/3 • The accessibility level of the result of a function call is that of the *master of the function call*, which is determined by the point of call as follows:

10.2/5 - If the result type at the point of the function (or access-to-function type) declaration is a composite type, and the result is used (in its entirety) to directly initialize part of an object, the master is that of the object being initialized. In the case where the initialized object is a coextension (see below) that becomes a coextension of another object, the master is that of the eventual object to which the coextension will be transferred.

10.3/5 - If the result is of an anonymous access type and is converted to a (named or anonymous) access type, the master is determined following the rules given below for determining the master of an object created by an allocator (even if the access result is of an access-to-subprogram type);

10.4/5 - *This paragraph was deleted.*

- If the call itself defines the result of a function *F*, or has an accessibility level that is tied to the result of such a function *F*, then the master of the call is that of the master of the call invoking *F*; 10.5/5
- In other cases, the master of the call is that of the innermost master that evaluates the function call. 10.6/3

In the case of a call to a function whose result type is an anonymous access type, the accessibility level of the type of the result of the function call is also determined by the point of call as described above. 10.7/3

- Within a return statement, the accessibility level of the return object is that of the execution of the return statement. If the return statement completes normally by returning from the function, then prior to leaving the function, the accessibility level of the return object changes to be a level determined by the point of call, as does the level of any coextensions (see below) of the return object. 10.8/3
- The accessibility level of a derived access type is the same as that of its ultimate ancestor. 11
- The accessibility level of the anonymous access type defined by an access_definition of an object_renaming_declaration is the same as that of the renamed view. 11.1/2
- The accessibility level of the anonymous access type defined by an access_definition of a loop_parameter_subtype_indication is that of the loop parameter. 11.2/5
- The accessibility level of the anonymous access type of an access discriminant in the subtype_indication or qualified_expression of an allocator, or in the expression or return_subtype_indication of a return statement is determined as follows: 12/2
  - If the value of the access discriminant is determined by a discriminant_association in a subtype_indication, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null); 12.1/2
  - If the value of the access discriminant is determined by a default_expression in the declaration of the discriminant, the level of the object or subprogram designated by the associated value (or library level if null); 12.2/3
  - If the value of the access discriminant is determined by a record_component_association in an aggregate, the accessibility level of the object or subprogram designated by the associated value (or library level if the value is null); 12.3/3
  - In other cases, where the value of the access discriminant is determined by an object with an unconstrained nominal subtype, the accessibility level of the object. 12.4/3
- The accessibility level of the anonymous access type of an access discriminant in any other context is that of the enclosing object. 12.5/3
- The accessibility level of the anonymous access type of an access parameter specifying an access-to-object type is the same as that of the view designated by the actual (or library-level if the actual is null). 13/3
- The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is deeper than that of any master; all such anonymous access types have this same level. 13.1/2
- The accessibility level of the anonymous access subtype defined by a return_subtype_indication that is an access_definition (see 6.5) is that of the result subtype of the enclosing function. 13.2/4
- The accessibility level of the type of a stand-alone object of an anonymous access-to-object type is the same as the accessibility level of the type of the access value most recently assigned to the object; accessibility checks ensure that this is never deeper than that of the declaration of the stand-alone object. 13.3/4

14/6 • The accessibility level of an object created by an allocator is the same as that of the access type, except for an allocator of an anonymous access type (an *anonymous allocator*) in certain contexts, as follows: For an anonymous allocator that defines the result of a function with an access result, the accessibility level is determined as though the allocator were in place of the call of the function; in the special case of a call that is the operand of a type conversion, the level is that of the target access type of the conversion. For an anonymous allocator defining the value of an access parameter, the accessibility level is that of the innermost master of the evaluation of the call. For an anonymous allocator whose type is that of a stand-alone object of an anonymous access-to-object type, the accessibility level is that of the declaration of the stand-alone object. For one defining an access discriminant, the accessibility level is determined as follows:

14.1/3 • for an allocator used to define the discriminant of an object, the level of the object;

14.2/3 • for an allocator used to define the constraint in a subtype_indication in any other context, the level of the master that elaborates the subtype_indication.

14.3/3 • *This paragraph was deleted.*

14.4/3 In the first case, the allocated object is said to be a *coextension* of the object whose discriminant designates it, as well as of any object of which the discriminated object is itself a coextension or subcomponent. If the allocated object is a coextension of an anonymous object representing the result of an aggregate or function call that is used (in its entirety) to directly initialize a part of an object, after the result is assigned, the coextension becomes a coextension of the object being initialized and is no longer considered a coextension of the anonymous object. All coextensions of an object (which have not thus been transfered by such an initialization) are finalized when the object is finalized (see 7.6.1).

14.5/3 • Within a return statement, the accessibility level of the anonymous access type of an access result is that of the master of the call.

15/3 • The accessibility level of a view of an object or subprogram designated by an access value is the same as that of the access type.

16 • The accessibility level of a component, protected subprogram, or entry of (a view of) a composite object is the same as that of (the view of) the composite object.

16.1/5 In the above rules, the operative constituents of a name or expression (see 4.4) are considered to be used in a given context if the enclosing name or expression is used in that context.

17 One accessibility level is defined to be *statically deeper* than another in the following cases:

18/5 • For a master construct that is statically nested within another master construct, the accessibility level of the inner master construct is statically deeper than that of the outer master construct.

18.1/2 • The accessibility level of the anonymous access type of an access parameter specifying an access-to-subprogram type is statically deeper than that of any master; all such anonymous access types have this same level.

19/5 • The statically deeper relationship does not apply to the accessibility level of the following:

19.1/5 • the anonymous type of an access parameter specifying an access-to-object type;

19.2/5 • the type of a stand-alone object of an anonymous access-to-object type;

19.3/5 • a raise_expression;

19.4/5 • a descendant of a generic formal type;

19.5/5 • a descendant of a type declared in a generic formal package.

19.6/5 • When the statically deeper relationship does not apply, the accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

- *This paragraph was deleted.* 19.7/5
- When within a function body or the return expression of an expression function, the accessibility level of the master representing an execution of the function is statically deeper than that of the master of the function call invoking that execution, independent of how the master of the function call is determined (see above). 19.8/5
- For determining whether one level is statically deeper than another when within a generic package body, the generic package is presumed to be instantiated at the same level as where it was declared; runtime checks are required in the case of more deeply nested instantiations. 20/5
- For determining whether one level is statically deeper than another when within the declarative region of a type_declaration, the current instance of the type is presumed to be an object created at a deeper level than that of the type. 21

Notwithstanding other rules given above, the accessibility level of an entity that is tied to that of an explicitly aliased formal parameter of an enclosing function is considered (both statically and dynamically) to be the same as that of an entity whose accessibility level is tied to that of the return object of that function. 21.1/5

The accessibility level of all library units is called the *library level*; a library-level declaration or entity is one whose accessibility level is the library level. 22

The following attribute is defined for a prefix X that denotes an aliased view of an object: 23

X'Access X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. X shall denote an aliased view of an object, including possibly the current instance (see 8.6) of a limited type within its definition, or a formal parameter or generic formal object of a tagged type. The view denoted by the prefix X shall satisfy the following additional requirements, presuming the expected type for X'Access is the general access type *A* with designated type *D*: 24/1

- If *A* is an access-to-variable type, then the view shall be a variable; on the other hand, if *A* is an access-to-constant type, the view may be either a constant or a variable. 25
- The view shall not be a subcomponent that depends on discriminants of an object unless the object is known to be constrained. 26/3
- If *A* is a named access type and *D* is a tagged type, then the type of the view shall be covered by *D*; if *A* is anonymous and *D* is tagged, then the type of the view shall be either *D*'Class or a type covered by *D*; if *D* is untagged, then the type of the view shall be *D*, and either: 27/2
  - the designated subtype of *A* shall statically match the nominal subtype of the view; or 27.1/2
  - *D* shall be discriminated in its full view and unconstrained in any partial view, and the designated subtype of *A* shall be unconstrained. 27.2/4
- The accessibility level of the view shall not be statically deeper than that of the access type *A*. 28/3

In addition to the places where Legality Rules normally apply (see 12.3), these requirements apply also in the private part of an instance of a generic unit. 28.1/3

A check is made that the accessibility level of X is not deeper than that of the access type *A*. If this check fails, Program_Error is raised. 29

30/5 If the nominal subtype of X does not statically match the designated subtype of *A*, a view conversion of X to the designated subtype is evaluated (which can raise Constraint_Error — see 4.6) and the value of X'Access designates that view.

31 The following attribute is defined for a prefix P that denotes a subprogram:

32/5 P'Access P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. The accessibility level of P shall not be statically deeper than that of *S*. If *S* is nonblocking, P shall be nonblocking. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. The profile of P shall be subtype conformant with the designated profile of *S*, and shall not be Intrinsic. If the subprogram denoted by P is declared within a generic unit, and the expression P'Access occurs within the body of that generic unit or within the body of a generic unit declared within the declarative region of the generic unit, then the ultimate ancestor of *S* shall be either a nonformal type declared within the generic unit or an anonymous access type of an access parameter.

Legality Rules

32.1/3 An expression is said to have *distributed accessibility* if it is

32.2/3 - a conditional_expression (see 4.5.7); or

32.3/5 - a declare_expression (see 4.5.9) whose *body*_expression has distributed accessibility; or

32.4/5 - a view conversion, qualified_expression, or parenthesized expression whose operand has distributed accessibility.

32.5/5 The statically deeper relationship does not apply to the accessibility level of an expression having distributed accessibility; that is, such an accessibility level is not considered to be statically deeper, nor statically shallower, than any other.

32.6/5 Any static accessibility requirement that is imposed on an expression that has distributed accessibility (or on its type) is instead imposed on the *dependent*_expressions of the underlying conditional_expression. This rule is applied recursively if a *dependent*_expression also has distributed accessibility.

33 NOTE 1 The Unchecked_Access attribute yields the same result as the Access attribute for objects, but has fewer restrictions (see 13.10). There are other predefined operations that yield access values: an allocator can be used to create an object, and return an access value that designates it (see 4.8); evaluating the literal **null** yields a null access value that designates no entity at all (see 4.2).

34/2 NOTE 2 The predefined operations of an access type also include the assignment operation, qualification, and membership tests. Explicit conversion is allowed between general access types with matching designated subtypes; explicit conversion is allowed between access-to-subprogram types with subtype conformant profiles (see 4.6). Named access types have predefined equality operators; anonymous access types do not, but they can use the predefined equality operators for *universal_access* (see 4.5.2).

35 NOTE 3 The object or subprogram designated by an access value can be named with a dereference, either an explicit_dereference or an implicit_dereference. See 4.1.

36 NOTE 4 A call through the dereference of an access-to-subprogram value is never a dispatching call.

37/5 NOTE 5 The Access attribute for subprograms and parameters of an anonymous access-to-subprogram type can be used together to implement "downward closures" — that is, to pass a more-nested subprogram as a parameter to a less-nested subprogram, as can be appropriate for an iterator abstraction or numerical integration. Downward closures can also be implemented using generic formal subprograms (see 12.6). Unlike for objects, there is no Unchecked_Access attribute for subprograms.

38/5 NOTE 6 Using an access-to-class-wide tagged type with a dispatching operation is a potentially more structured alternative to using an access-to-subprogram type.

NOTE 7 An implementation can consider two access-to-subprogram values to be unequal, even though they designate the same subprogram. For instance, this can happen because one points directly to the subprogram, while the other points to a special prologue that performs an Elaboration_Check and then jumps to the subprogram. See 4.5.2. 39/5

Examples

Example of use of the Access attribute: 40

```
Becky : Person_Name := new Person(F);        -- see 3.10.1
Cars  : array (1..2) of aliased Car;
   ...
Becky.Vehicle := Cars(1)'Access;
Casey.Vehicle := Cars(2)'Access;
```
41/5

3.11 Declarative Parts

A declarative_part contains declarative_items (possibly none). 1

Syntax

declarative_part ::= {declarative_item} 2

declarative_item ::= 3
  basic_declarative_item | body

basic_declarative_item ::= 4/1
  basic_declaration | aspect_clause | use_clause

body ::= proper_body | body_stub 5

proper_body ::= 6
  subprogram_body | package_body | task_body | protected_body

Static Semantics

The list of declarative_items of a declarative_part is called the *declaration list* of the declarative_part. 6.1/2

Dynamic Semantics

The elaboration of a declarative_part consists of the elaboration of the declarative_items, if any, in the order in which they are given in the declarative_part. 7

An elaborable construct is in the *elaborated* state after the normal completion of its elaboration. Prior to that, it is *not yet elaborated*. 8

For a construct that attempts to use a body, a check (Elaboration_Check) is performed, as follows: 9

- For a call to a (non-protected) subprogram that has an explicit body, a check is made that the body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order. 10/1
- For a call to a protected operation of a protected type (that has a body — no check is performed if the protected type is imported — see B.1), a check is made that the protected_body is already elaborated. This check and the evaluations of any actual parameters of the call are done in an arbitrary order. 11/3
- For the activation of a task, a check is made by the activator that the task_body is already elaborated. If two or more tasks are being activated together (see 9.2), as the result of the elaboration of a declarative_part or the initialization for the object created by an allocator, this check is done for all of them before activating any of them. 12

13 • For the instantiation of a generic unit that has a body, a check is made that this body is already elaborated. This check and the evaluation of any explicit_generic_actual_parameters of the instantiation are done in an arbitrary order.

14 The exception Program_Error is raised if any of these checks fails.

3.11.1 Completions of Declarations

1/3 Declarations sometimes come in two parts. A declaration that requires a second part is said to *require completion*. The second part is called the *completion* of the declaration (and of the entity declared), and is either another declaration, a body, or a pragma. A *body* is a body, an entry_body, a null_procedure_declaration or an expression_function_declaration that completes another declaration, or a renaming-as-body (see 8.5.4).

Name Resolution Rules

2 A construct that can be a completion is interpreted as the completion of a prior declaration only if:

3 • The declaration and the completion occur immediately within the same declarative region;

4 • The defining name or defining_program_unit_name in the completion is the same as in the declaration, or in the case of a pragma, the pragma applies to the declaration;

5 • If the declaration is overloadable, then the completion either has a type-conformant profile, or is a pragma.

Legality Rules

6/3 An implicit declaration shall not have a completion. For any explicit declaration that is specified to *require completion*, there shall be a corresponding explicit completion, unless the declared entity is imported (see B.1).

7 At most one completion is allowed for a given declaration. Additional requirements on completions appear where each kind of completion is defined.

8 A type is *completely defined* at a place that is after its full type definition (if it has one) and after all of its subcomponent types are completely defined. A type shall be completely defined before it is frozen (see 13.14 and 7.3).

9/3 NOTE 1 Completions are in principle allowed for any kind of explicit declaration. However, for some kinds of declaration, the only allowed completion is an implementation-defined pragma, and implementations are not required to have any such pragmas.

10/5 NOTE 2 There are rules that prevent premature uses of declarations that have a corresponding completion. The Elaboration_Checks of 3.11 prevent such uses at run time for subprograms, protected operations, tasks, and generic units. The freezing rules (see 13.14) prevent, at compile time, premature uses of other entities such as private types and deferred constants.

4 Names and Expressions

The rules applicable to the different forms of name and expression, and to their evaluation, are given in this clause. 1/3

4.1 Names

Names can denote declared entities, whether declared explicitly or implicitly (see 3.1). Names can also denote objects or subprograms designated by access values; the results of type_conversions or function_calls; subcomponents and slices of objects and values; protected subprograms, single entries, entry families, and entries in families of entries. Finally, names can denote attributes of any of the foregoing. 1

Syntax

name ::= 2/5
    direct_name | explicit_dereference
  | indexed_component | slice
  | selected_component | attribute_reference
  | type_conversion | function_call
  | character_literal | qualified_expression
  | generalized_reference | generalized_indexing
  | target_name

direct_name ::= identifier | operator_symbol 3

prefix ::= name | implicit_dereference 4

explicit_dereference ::= name.**all** 5

implicit_dereference ::= name 6

Certain forms of name (indexed_components, selected_components, slices, and attribute_references) include a prefix that is either itself a name that denotes some related entity, or an implicit_dereference of an access value that designates some related entity. 7/3

Name Resolution Rules

The name in a *dereference* (either an implicit_dereference or an explicit_dereference) is expected to be of any access type. 8

Static Semantics

If the type of the name in a dereference is some access-to-object type *T*, then the dereference denotes a view of an object, the *nominal subtype* of the view being the designated subtype of *T*. If the designated subtype has unconstrained discriminants, the (actual) subtype of the view is constrained by the values of the discriminants of the designated object, except when there is a partial view of the type of the designated subtype that does not have discriminants, in which case the dereference is not constrained by its discriminant values. 9/3

If the type of the name in a dereference is some access-to-subprogram type *S*, then the dereference denotes a view of a subprogram, the *profile* of the view being the designated profile of *S*. 10

Dynamic Semantics

11/2 The evaluation of a name determines the entity denoted by the name. This evaluation has no other effect for a name that is a direct_name or a character_literal.

12 The evaluation of a name that has a prefix includes the evaluation of the prefix. The evaluation of a prefix consists of the evaluation of the name or the implicit_dereference. The prefix denotes the entity denoted by the name or the implicit_dereference.

13 The evaluation of a dereference consists of the evaluation of the name and the determination of the object or subprogram that is designated by the value of the name. A check is made that the value of the name is not the null access value. Constraint_Error is raised if this check fails. The dereference denotes the object or subprogram designated by the value of the name.

Examples

14 *Examples of direct names:*

15
```
Pi       -- the direct name of a number              (see 3.3.2)
Limit    -- the direct name of a constant            (see 3.3.1)
Count    -- the direct name of a scalar variable     (see 3.3.1)
Board    -- the direct name of an array variable     (see 3.6.1)
Matrix   -- the direct name of a type                (see 3.6)
Random   -- the direct name of a function            (see 6.1)
Error    -- the direct name of an exception          (see 11.1)
```

16 *Examples of dereferences:*

17
```
Next_Car.all     -- explicit dereference denoting the object designated
                 -- by the access variable Next_Car (see 3.10.1)
Next_Car.Owner   -- selected component with implicit dereference;
                 -- same as Next_Car.all.Owner
```

4.1.1 Indexed Components

1 An indexed_component denotes either a component of an array or an entry in a family of entries.

Syntax

2 indexed_component ::= prefix(expression {, expression})

Name Resolution Rules

3 The prefix of an indexed_component with a given number of expressions shall resolve to denote an array (after any implicit dereference) with the corresponding number of index positions, or shall resolve to denote an entry family of a task or protected object (in which case there shall be only one expression).

4 The expected type for each expression is the corresponding index type.

Static Semantics

5 When the prefix denotes an array, the indexed_component denotes the component of the array with the specified index value(s). The nominal subtype of the indexed_component is the component subtype of the array type.

6 When the prefix denotes an entry family, the indexed_component denotes the individual entry of the entry family with the specified index value.

Dynamic Semantics

For the evaluation of an indexed_component, the prefix and the expressions are evaluated in an arbitrary order. The value of each expression is converted to the corresponding index type. A check is made that each index value belongs to the corresponding index range of the array or entry family denoted by the prefix. Constraint_Error is raised if this check fails. 7

Examples

Examples of indexed components: 8

```
My_Schedule(Sat)    -- a component of a one-dimensional array   (see 3.6.1)
Page(10)            -- a component of a one-dimensional array   (see 3.6)
Board(M, J + 1)     -- a component of a two-dimensional array   (see 3.6.1)
Page(10)(20)        -- a component of a component               (see 3.6)
Request(Medium)     -- an entry in a family of entries          (see 9.1)
Next_Frame(L)(M, N) -- a component of a function call           (see 6.1)
```
9

Distinct notations are used for components of multidimensional arrays (such as Board) and arrays of arrays (such as Page). The components of an array of arrays are arrays and can therefore be indexed. Thus Page(10)(20) denotes the 20th component of Page(10). In the last example Next_Frame(L) is a function call returning an access value that designates a two-dimensional array. 10/5

4.1.2 Slices

A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array. A slice of a variable is a variable; a slice of a constant is a constant; a slice of a value is a value. 1

Syntax

slice ::= prefix(discrete_range) 2

Name Resolution Rules

The prefix of a slice shall resolve to denote a one-dimensional array (after any implicit dereference). 3

The expected type for the discrete_range of a slice is the index type of the array type. 4

Static Semantics

A slice denotes a one-dimensional array formed by the sequence of consecutive components of the array denoted by the prefix, corresponding to the range of values of the index given by the discrete_range. 5

The type of the slice is that of the prefix. Its bounds are those defined by the discrete_range. 6

Dynamic Semantics

For the evaluation of a slice, the prefix and the discrete_range are evaluated in an arbitrary order. If the slice is not a *null slice* (a slice where the discrete_range is a null range), then a check is made that the bounds of the discrete_range belong to the index range of the array denoted by the prefix. Constraint_Error is raised if this check fails. 7

NOTE 1 By the rules given in 3.10.2, a slice is illegal as the prefix of an Access attribute_reference, even if the components or the array as a whole are aliased. 8/5

NOTE 2 For a one-dimensional array A, the slice A(N .. N) denotes an array that has only one component; its type is the type of A. On the other hand, A(N) denotes a component of the array A and has the corresponding component type. 9

Examples

10 *Examples of slices:*

11
```
Stars(1 .. 15)          -- a slice of 15 characters          (see 3.6.3)
Page(10 .. 10 + Size)   -- a slice of 1 + Size components    (see 3.6)
Page(L)(A .. B)         -- a slice of the array Page(L)      (see 3.6)
Stars(1 .. 0)           -- a null slice                      (see 3.6.3)
My_Schedule(Weekday)    -- bounds given by subtype           (see 3.6.1 and 3.5.1)
Stars(5 .. 15)(K)       -- same as Stars(K)                  (see 3.6.3)
                        -- provided that K is in 5 .. 15
```

4.1.3 Selected Components

1 Selected_components are used to denote components (including discriminants), entries, entry families, and protected subprograms; they are also used as expanded names as described below.

Syntax

2 selected_component ::= prefix . selector_name

3 selector_name ::= identifier | character_literal | operator_symbol

Name Resolution Rules

4 A selected_component is called an *expanded name* if, according to the visibility rules, at least one possible interpretation of its prefix denotes a package or an enclosing named construct (directly, not through a subprogram_renaming_declaration or generic_renaming_declaration).

5 A selected_component that is not an expanded name shall resolve to denote one of the following:

6 • A component (including a discriminant):

7 The prefix shall resolve to denote an object or value of some non-array composite type (after any implicit dereference). The selector_name shall resolve to denote a discriminant_specification of the type, or, unless the type is a protected type, a component_declaration of the type. The selected_component denotes the corresponding component of the object or value.

8 • A single entry, an entry family, or a protected subprogram:

9 The prefix shall resolve to denote an object or value of some task or protected type (after any implicit dereference). The selector_name shall resolve to denote an entry_declaration or subprogram_declaration occurring (implicitly or explicitly) within the visible part of that type. The selected_component denotes the corresponding entry, entry family, or protected subprogram.

9.1/2 • A view of a subprogram whose first formal parameter is of a tagged type or is an access parameter whose designated type is tagged:

9.2/6 The prefix (after any implicit dereference) shall resolve to denote an object or value of a specific tagged type *T* or class-wide type *T*'Class. The selector_name shall resolve to denote a view of a subprogram declared immediately within the declarative region in which an ancestor of the type *T* is declared. The first formal parameter of the subprogram shall be of type *T*, or a class-wide type that covers *T*, or an access parameter designating one of these types. The designator of the subprogram shall not be the same as that of a component of the tagged type visible at the point of the selected_component. The subprogram shall not be an implicitly declared primitive operation of type *T* that overrides an inherited subprogram implemented by an entry or protected subprogram visible at the point of the selected_component. The selected_component denotes a view of this subprogram that omits the first formal parameter. This view is called a *prefixed view* of the subprogram, and the prefix of the selected_component (after any implicit dereference) is called the *prefix* of the prefixed view. In the special case where the prefix is

class-wide and controlling, the prefixed view is not abstract, even if the unprefixed view is abstract; otherwise the prefixed view is abstract if and only if the unprefixed view is abstract.

In the context of a subprogram renaming (see 8.5.4) or the matching of an actual subprogram with a formal subprogram (see 12.6), a prefixed view whose prefix is class-wide and controlling shall have an expected profile such that each formal parameter or result type corresponding to a controlling parameter (or access parameter) or controlling result (or access result) in the unprefixed view, is (or designates) the same class-wide type as the prefix. In a renaming or matching of a prefixed view whose prefix is of a specific type or is not controlling, the unprefixed view shall not be abstract and the types of the formal parameters or result in the expected profile shall match those of the corresponding formal parameters or result of the unprefixed view. In either case, any constraints, null exclusions, or predicates on the formal parameters or results in the view declared by the renaming or instantiation come from the unprefixed view, as in the normal case. When renaming or matching a prefixed view whose prefix is tag-indeterminate and controlling, the controlling tag is statically determined to be that of the type of the prefix. 9.3/6

An expanded name shall resolve to denote a declaration that occurs immediately within a named declarative region, as follows: 10

- The prefix shall resolve to denote either a package (including the current instance of a generic package, or a rename of a package), or an enclosing named construct. 11
- The selector_name shall resolve to denote a declaration that occurs immediately within the declarative region of the package or enclosing construct (the declaration shall be visible at the place of the expanded name — see 8.3). The expanded name denotes that declaration. 12
- If the prefix does not denote a package, then it shall be a direct_name or an expanded name, and it shall resolve to denote a program unit (other than a package), the current instance of a type, a block_statement, a loop_statement, or an accept_statement (in the case of an accept_statement or entry_body, no family index is allowed); the expanded name shall occur within the declarative region of this construct. Further, if this construct is a callable construct and the prefix denotes more than one such enclosing callable construct, then the expanded name is ambiguous, independently of the selector_name. 13

Legality Rules

For a prefixed view of a subprogram whose first formal parameter is an access parameter, the prefix shall be legal as the prefix of an attribute_reference with attribute_designator Access appearing as the first actual parameter in a call on the unprefixed view of the subprogram. 13.1/5

For a subprogram whose first parameter is of mode **in out** or **out**, or of an anonymous access-to-variable type, the prefix of any prefixed view shall denote a variable. 13.2/2

Dynamic Semantics

The evaluation of a selected_component includes the evaluation of the prefix. 14

For a selected_component that denotes a component of a variant, a check is made that the values of the discriminants are such that the value or object denoted by the prefix has this component. The exception Constraint_Error is raised if this check fails. 15

Examples

16 *Examples of selected components:*

17/5
```
Tomorrow.Month       -- a record component                    (see 3.8)
Next_Car.Owner       -- a record component                    (see 3.10.1)
Next_Car.Owner.Age   -- a record component                    (see 3.10.1)
                     -- the previous two lines involve implicit dereferences
Writer.Unit          -- a record component (a discriminant)   (see 3.8.1)
Min_Cell(H).Value    -- a record component of the result      (see 6.1)
                     -- of the function call Min_Cell(H)
Cashier.Append       -- a prefixed view of a procedure        (see 3.9.4)
Control.Seize        -- an entry of a protected object        (see 9.4)
Pool(K).Write        -- an entry of the task Pool(K)          (see 9.1)
```

18 *Examples of expanded names:*

19
```
Key_Manager."<"      -- an operator of the visible part of a package   (see 7.3.1)
Dot_Product.Sum      -- a variable declared in a function body         (see 6.1)
Buffer.Pool          -- a variable declared in a protected unit        (see 9.11)
Buffer.Read          -- an entry of a protected unit                   (see 9.11)
Swap.Temp            -- a variable declared in a block statement       (see 5.6)
Standard.Boolean     -- the name of a predefined type                  (see A.1)
```

4.1.4 Attributes

1 An *attribute* is a characteristic of an entity that can be queried via an attribute_reference or a range_-attribute_reference.

Syntax

2/5 attribute_reference ::=
prefix'attribute_designator
| reduction_attribute_reference

3/2 attribute_designator ::=
identifier[(*static*_expression)]
| Access | Delta | Digits | Mod

4 range_attribute_reference ::= prefix'range_attribute_designator

5 range_attribute_designator ::= Range[(*static*_expression)]

Name Resolution Rules

6/6 In an attribute_reference that is not a reduction_attribute_reference, if the attribute_designator is for an attribute defined for (at least some) objects of an access type, then the prefix is never interpreted as an implicit_dereference; otherwise (and for all range_attribute_references and reduction_attribute_-references), if there is a prefix and the type of the name within the prefix is of an access type, the prefix is interpreted as an implicit_dereference. Similarly, if the attribute_designator is for an attribute defined for (at least some) functions, then the prefix is never interpreted as a parameterless function_call; otherwise (and for all range_attribute_references and reduction_attribute_references), if there is a prefix and the prefix consists of a name that denotes a function, it is interpreted as a parameterless function_call. Any requirement that a given attribute_designator may only be referenced from within a construct denoted by the prefix is considered a Name Resolution Rule for the prefix.

7 The expression, if any, in an attribute_designator or range_attribute_designator is expected to be of any integer type.

Legality Rules

The expression, if any, in an attribute_designator or range_attribute_designator shall be static. 8

Static Semantics

An attribute_reference denotes a value, an object, a subprogram, or some other kind of program entity. Unless explicitly specified otherwise, for an attribute_reference that denotes a value or an object, if its type is scalar, then its nominal subtype is the base subtype of the type; if its type is tagged, its nominal subtype is the first subtype of the type; otherwise, its nominal subtype is a subtype of the type without any constraint, null_exclusion, or predicate. Similarly, unless explicitly specified otherwise, for an attribute_reference that denotes a function, when its result type is scalar, its result subtype is the base subtype of the type, when its result type is tagged, the result subtype is the first subtype of the type, and when the result type is some other type, the result subtype is a subtype of the type without any constraint, null_exclusion, or predicate. 9/4

A range_attribute_reference X'Range(N) is equivalent to the range X'First(N) .. X'Last(N), except that the prefix is only evaluated once. Similarly, X'Range is equivalent to X'First .. X'Last, except that the prefix is only evaluated once. 10

Dynamic Semantics

The evaluation of a range_attribute_reference or an attribute_reference that is not a reduction_attribute_reference consists of the evaluation of the prefix. The evaluation of a reduction_attribute_reference is defined in 4.5.10. 11/5

Implementation Permissions

An implementation may provide implementation-defined attributes; the identifier for such an implementation-defined attribute shall differ from those of the language-defined attributes. 12/5

An implementation may extend the definition of a language-defined attribute by accepting uses of that attribute that would otherwise be illegal in the following cases: 13/5

- in order to support compatibility with a previous edition of of this Reference Manual; or 14/5
- in the case of a language-defined attribute whose prefix is required by this document to be a floating point subtype, an implementation may accept an attribute_reference whose prefix is a fixed point subtype; the semantics of such an attribute_reference are implementation defined. 15/5

NOTE 1 Attributes are defined throughout this document, and are summarized in K.2. 16

NOTE 2 By the general rules given above, there is no expected type or profile for the name in a prefix of an attribute_reference (or a range_attribute_reference), which means that no context can be used to resolve the name. However, by the rules given in 3.10.2 for the case of the Access attribute, the expected type for the attribute_reference will be a single access type, and the resolution of the name can make use of the designated type or profile of this access type. 17/5

Examples

Examples of attributes: 18

```
Color'First          -- minimum value of the enumeration type Color  (see 3.5.1)
Rainbow'Base'First   -- same as Color'First                          (see 3.5.1)
Real'Digits          -- precision of the type Real                   (see 3.5.7)
Board'Last(2)        -- upper bound of the second dimension of Board (see 3.6.1)
Board'Range(1)       -- index range of the first dimension of Board  (see 3.6.1)
Pool(K)'Terminated   -- True if task Pool(K) is terminated           (see 9.1)
Date'Size            -- number of bits for records of type Date      (see 3.8)
Message'Address      -- address of the record variable Message
                     --                                              (see 3.7.1)
```
19

4.1.5 User-Defined References

Static Semantics

1/3 Given a discriminated type *T*, the following type-related operational aspect may be specified:

2/3 Implicit_Dereference
: This aspect is specified by a name that denotes an access discriminant declared for the type *T*.

3/3 A (view of a) type with a specified Implicit_Dereference aspect is a *reference type*. A *reference object* is an object of a reference type. The discriminant named by the Implicit_Dereference aspect is the *reference discriminant* of the reference type or reference object. A generalized_reference is a name that identifies a reference object, and denotes the object or subprogram designated by the reference discriminant of the reference object.

Syntax

4/3 generalized_reference ::= *reference_object_*name

Name Resolution Rules

5/3 The expected type for the *reference_object_*name in a generalized_reference is any reference type.

Static Semantics

5.1/4 The Implicit_Dereference aspect is nonoverridable (see 13.1.1).

6/3 A generalized_reference denotes a view equivalent to that of a dereference of the reference discriminant of the reference object.

7/5 Given a reference type *T*, the Implicit_Dereference aspect is inherited by descendants of type *T* if not overridden (which is only permitted if confirming). If a descendant type constrains the value of the reference discriminant of *T* by a new discriminant, that new discriminant is the reference discriminant of the descendant. If the descendant type constrains the value of the reference discriminant of *T* by an expression other than the name of a new discriminant, a generalized_reference that identifies an object of the descendant type denotes the object or subprogram designated by the value of this constraining expression.

Dynamic Semantics

8/3 The evaluation of a generalized_reference consists of the evaluation of the *reference_object_*name and a determination of the object or subprogram designated by the reference discriminant of the named reference object. A check is made that the value of the reference discriminant is not the null access value. Constraint_Error is raised if this check fails. The generalized_reference denotes the object or subprogram designated by the value of the reference discriminant of the named reference object.

Examples

9/5 *Examples of the specification and use of generalized references:*

10/3
```
type Barrel is tagged ...  -- holds objects of type Element
```

11/3
```
type Ref_Element(Data : access Element) is limited private
   with Implicit_Dereference => Data;
      -- This Ref_Element type is a "reference" type.
      -- "Data" is its reference discriminant.
```

```
function Find (B : aliased in out Barrel; Key : String)
   return Ref_Element;
   -- Returns a reference to an element of a barrel.
B: aliased Barrel;
...
Find (B, "grape") := Element'(...);  -- Assign through a reference.
-- This is equivalent to:
Find (B, "grape").Data.all := Element'(...);
```

Usage

For most uses, and in particular for types used with user-defined indexing (see 4.1.6), reference types should be limited types. Doing so helps avoid ambiguities that otherwise can arise. Ambiguities are possible because a name that can be interpreted as a generalized_reference can also be interpreted in the usual way (that is, as a name denoting an object of the reference type). For example, if X and Y are objects of a reference type T1 that has an "**access** T2" reference discriminant, then the expression X = Y would be ambiguous if equality operators for both T1 and T2 are directly visible. See the example below where T1 is Bad_Ref and T2 is Barrel. Such ambiguities can especially complicate the use of user-defined indexing.

```
type Bad_Ref (D : access Barrel) is null record
    with Implicit_Dereference => D;
function Create_Ref (B : aliased in out Barrel) return Bad_Ref;
X, Y : Bad_Ref := Create_Ref (B);
   ...
if X = Y then -- Ambiguous, "=" can be for type Bad_Ref or type Barrel.
   ...
```

4.1.6 User-Defined Indexing

Static Semantics

Given a tagged type *T*, the following type-related, operational aspects may be specified:

Constant_Indexing

This aspect shall be specified by a name that denotes one or more functions declared immediately within the same declaration list in which *T*, or the declaration completed by *T*, is declared. All such functions shall have at least two parameters, the first of which is of type *T* or *T*'Class, or is an access-to-constant parameter with designated type *T* or *T*'Class.

Variable_Indexing

This aspect shall be specified by a name that denotes one or more functions declared immediately within the same declaration list in which *T*, or the declaration completed by *T*, is declared. All such functions shall have at least two parameters, the first of which is of type *T* or *T*'Class, or is an access parameter with designated type *T* or *T*'Class. All such functions shall have a return type that is a reference type (see 4.1.5), whose reference discriminant is of an access-to-variable type.

These aspects are inherited by descendants of *T* (including the class-wide type *T*'Class).

An *indexable container type* is (a view of) a tagged type with at least one of the aspects Constant_Indexing or Variable_Indexing specified. An *indexable container object* is an object of an indexable container type. A generalized_indexing is a name that denotes the result of calling a function named by a Constant_Indexing or Variable_Indexing aspect.

The Constant_Indexing and Variable_Indexing aspects are nonoverridable (see 13.1.1).

Legality Rules

6/5 If an ancestor of a type *T* is an indexable container type, then any explicit specification of the Constant_Indexing or Variable_Indexing aspects shall be confirming; that is, the specified name shall match the inherited aspect (see 13.1.1).

Paragraphs 7 through 8 were deleted.

9/5 In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

9.1/5 A generalized_indexing is illegal if the equivalent prefixed view (see below) is illegal.

Syntax

10/3 generalized_indexing ::= *indexable_container_object_*prefix actual_parameter_part

Name Resolution Rules

11/3 The expected type for the *indexable_container_object_*prefix of a generalized_indexing is any indexable container type.

12/3 If the Constant_Indexing aspect is specified for the type of the *indexable_container_object_*prefix of a generalized_indexing, then the generalized_indexing is interpreted as a *constant indexing* under the following circumstances:

13/3 - when the Variable_Indexing aspect is not specified for the type of the *indexable_container_object_*prefix;

14/3 - when the *indexable_container_object_*prefix denotes a constant;

15/3 - when the generalized_indexing is used within a primary where a name denoting a constant is permitted.

16/3 Otherwise, the generalized_indexing is interpreted as a *variable indexing*.

17/3 When a generalized_indexing is interpreted as a constant (or variable) indexing, it is equivalent to a call on a prefixed view of one of the functions named by the Constant_Indexing (or Variable_Indexing) aspect of the type of the *indexable_container_object_*prefix with the given actual_parameter_part, and with the *indexable_container_object_*prefix as the prefix of the prefixed view.

18/4 NOTE The Constant_Indexing and Variable_Indexing aspects cannot be redefined when inherited for a derived type, but the functions that they denote can be modified by overriding or overloading.

Examples

19/5 *Examples of the specification and use of generalized indexing:*

20/3
```
type Indexed_Barrel is tagged ...
  with Variable_Indexing => Find;
   -- Indexed_Barrel is an indexable container type,
   -- Find is the generalized indexing operation.
```

21/3
```
function Find (B : aliased in out Indexed_Barrel; Key : String)
   return Ref_Element;
   -- Return a reference to an element of a barrel (see 4.1.5).
```

22/3
```
IB: aliased Indexed_Barrel;
```

23/3
```
-- All of the following calls are then equivalent:
Find (IB,"pear").Data.all := Element'(...); -- Traditional call
IB.Find ("pear").Data.all := Element'(...); -- Call of prefixed view
IB.Find ("pear")          := Element'(...); -- Implicit dereference (see 4.1.5)
IB      ("pear")          := Element'(...); -- Implicit indexing and dereference
IB      ("pear").Data.all := Element'(...); -- Implicit indexing only
```

4.2 Literals

A *literal* represents a value literally, that is, by means of notation suited to its kind. A literal is either a numeric_literal, a character_literal, the literal **null**, or a string_literal. 1

Name Resolution Rules

This paragraph was deleted. 2/2

For a name that consists of a character_literal, either its expected type shall be a single character type, in which case it is interpreted as a parameterless function_call that yields the corresponding value of the character type, or its expected profile shall correspond to a parameterless function with a character result type, in which case it is interpreted as the name of the corresponding parameterless function declared as part of the character type's definition (see 3.5.1). In either case, the character_literal denotes the enumeration_literal_specification. 3

The expected type for a primary that is a string_literal shall be a single string type or a descendant of a type with a specified String_Literal aspect (see 4.2.1). In either case, the string_literal is interpreted to be of its expected type. If the expected type of an integer literal is a descendant of a type with a specified Integer_Literal aspect (see 4.2.1), the literal is interpreted to be of its expected type; otherwise it is interpreted to be of type *universal_integer*. If the expected type of a real literal is a descendant of a type with a specified Real_Literal aspect (see 4.2.1), it is interpreted to be of its expected type; otherwise, it is interpreted to be of type *universal_real*. 4/6

Legality Rules

A character_literal that is a name shall correspond to a defining_character_literal of the expected type, or of the result type of the expected profile. 5

If the expected type for a string_literal is a string type, then for each character of the string_literal there shall be a corresponding defining_character_literal of the component type of the expected string type. 6/5

This paragraph was deleted. 7/2

Static Semantics

The literal **null** is of type *universal_access*. 8/5

Dynamic Semantics

If its expected type is a numeric type, the evaluation of a numeric literal yields the represented value. In other cases, the effect of evaluating a numeric literal is determined by the Integer_Literal or Real_Literal aspect that applies (see 4.2.1). 9/5

The evaluation of the literal **null** yields the null value of the expected type. 9.1/5

The evaluation of a string_literal that is a primary and has an expected type that is a string type, yields an array value containing the value of each character of the sequence of characters of the string_literal, as defined in 2.6. The bounds of this array value are determined according to the rules for positional_array_aggregates without an **others** choice (see 4.3.3). In other cases, the effect of evaluating a string_literal is determined by the String_Literal aspect that applies (see 4.2.1). 10/6

For the evaluation of a string_literal of a string type *T*, a check is made that the value of each character of the string_literal belongs to the component subtype of *T*. The exception Constraint_Error is raised if this check fails. 11/6

12 NOTE Enumeration literals that are identifiers rather than character_literals follow the normal rules for identifiers when used in a name (see 4.1 and 4.1.3). Character_literals used as selector_names follow the normal rules for expanded names (see 4.1.3).

Examples

13 *Examples of literals:*

14
```
3.14159_26536       -- a real literal
1_345               -- an integer literal
'A'                 -- a character literal
"Some Text"         -- a string literal
```

4.2.1 User-Defined Literals

1/5 Using one or more of the aspects defined below, a type may be specified to allow the use of one or more kinds of literals as values of the type.

Static Semantics

2/5 The following type-related operational aspects (collectively known as *user-defined literal aspects*) may be specified for a type *T*:

3/5 Integer_Literal
: This aspect is specified by a *function*_name that statically denotes a function with a result type of *T* and one **in** parameter that is of type String and is not explicitly aliased.

4/5 Real_Literal
: This aspect is specified by a *function*_name that statically denotes a function with a result type of *T* and one **in** parameter that is of type String and is not explicitly aliased, and optionally a second function (overloading the first) with a result type of *T* and two **in** parameters of type String that are not explicitly aliased.

5/5 String_Literal
: This aspect is specified by a *function*_name that statically denotes a function with a result type of *T* and one **in** parameter that is of type Wide_Wide_String and is not explicitly aliased.

6/5 User-defined literal aspects are nonoverridable (see 13.1.1).

7/5 When a numeric literal is interpreted as a value of a non-numeric type *T* or a string_literal is interpreted as a value of a type *T* that is not a string type (see 4.2), it is equivalent to a call to the subprogram denoted by the corresponding aspect of *T*: the Integer_Literal aspect for an integer literal, the Real_Literal aspect for a real literal, and the String_Literal aspect for a string_literal. The actual parameter of this notional call is a string_literal representing a sequence of characters that is the same as the sequence of characters in the original numeric literal, or the sequence represented by the original string literal.

8/5 Such a literal is said to be a *user-defined literal*.

9/5 When a named number that denotes a value of type *universal_integer* is interpreted as a value of a non-numeric type *T*, it is equivalent to a call to the function denoted by the Integer_Literal aspect of *T*. The actual parameter of this notional call is a String having a textual representation of a decimal integer literal optionally preceded by a minus sign, representing the same value as the named number.

10/5 When a named number that denotes a value of type *universal_real* is interpreted as a value of a non-numeric type *T*, it is equivalent to a call to the two-parameter function denoted by the Real_Literal aspect of *T*, if any. The actual parameters of this notional call are each a String with the textual representation of a decimal integer literal, with the first optionally preceded by a minus sign, where the first String

represents the same value as the numerator, and the second the same value as the denominator, of the named number when represented as a rational number in lowest terms, with a positive denominator.

Legality Rules

The Integer_Literal or Real_Literal aspect shall not be specified for a type *T* if the full view of *T* is a numeric type. The String_Literal aspect shall not be specified for a type *T* if the full view of *T* is a string type. 11/5

For a nonabstract type, the function directly specified for a user-defined literal aspect shall not be abstract. 12/5

For a tagged type with a partial view, a user-defined literal aspect shall not be directly specified on the full type. 13/5

This paragraph was deleted. 14/6

If a named number that denotes a value of type *universal_integer* is interpreted as a value of a non-numeric type *T*, *T* shall have an Integer_Literal aspect. If a named number that denotes a value of type *universal_real* is interpreted as a value of a non-numeric type *T*, *T* shall have a Real_Literal aspect, and the aspect shall denote a function that has two **in** parameters, both of type String, with result of type *T*. 15/5

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 16/5

Bounded (Run-Time) Errors

It is a bounded error if the evaluation of a literal or named number that has an expected type with a specified user-defined literal aspect propagates an exception. Either Program_Error or the exception propagated by the evaluation is raised at the point of use of the value of the literal or named number. If it is recognized prior to run time that evaluation of such a literal or named number will inevitably (if executed) result in such a bounded error, then this may be reported as an error prior to run time. 17/5

Examples

Examples of the specification and use of user-defined literals: 18/5

```
subtype Roman_Character is Wide_Wide_Character                                  19/5
   with Static_Predicate =>
      Roman_Character in 'I' | 'V' | 'X' | 'L' | 'C' | 'D' | 'M';

Max_Roman_Number : constant := 3_999;  -- MMMCMXCIX                             20/5

type Roman_Number is range 1 .. Max_Roman_Number                                21/5
   with String_Literal => To_Roman_Number;

function To_Roman_Number (S : Wide_Wide_String) return Roman_Number              22/5
   with Pre => S'Length > 0 and then
      (for all Char of S => Char in Roman_Character);

function To_Roman_Number (S : Wide_Wide_String) return Roman_Number is          23/5
   (declare
      R : constant array (Integer range <>) of Roman_Number :=
         (for D in S'Range => Roman_Digit'Enum_Rep
             (Roman_Digit'Wide_Wide_Value ('''' & S(D) & ''')));
                   -- See 3.5.2 and 13.4
    begin
      [for I in R'Range =>
         (if I < R'Last and then R(I) < R(I + 1) then -1 else 1) * R(I)]
            'Reduce("+", 0)
   );

X : Roman_Number := "III" * "IV" * "XII"; -- 144 (that is, CXLIV)               24/5
```

4.3 Aggregates

1 An *aggregate* combines component values into a composite value of an array type, record type, or record extension.

Syntax

2/5
```
aggregate ::=
   record_aggregate | extension_aggregate | array_aggregate
 | delta_aggregate | container_aggregate
```

Name Resolution Rules

3/6 The expected type for an aggregate shall be a single array type, a single type with the Aggregate aspect specified (or a class-wide type rooted at such a type), or a single descendant of a record type or of a record extension.

Legality Rules

4/5 A record_aggregate or extension_aggregate shall not be of a class-wide type.

Dynamic Semantics

5 For the evaluation of an aggregate, an anonymous object is created and values for the components or ancestor part are obtained (as described in the subsequent subclause for each kind of the aggregate) and assigned into the corresponding components or ancestor part of the anonymous object. Obtaining the values and the assignments occur in an arbitrary order. The value of the aggregate is the value of this object.

6/6 If an aggregate is of a tagged type, a check is made that its value belongs to the first subtype of the type. Constraint_Error is raised if this check fails. Any discriminant check is performed before the initialization of any nondiscriminant component of the aggregate object.

4.3.1 Record Aggregates

1 In a record_aggregate, a value is specified for each component of the record or record extension value, using either a named or a positional association.

Syntax

2
```
record_aggregate ::= (record_component_association_list)
```

3
```
record_component_association_list ::=
   record_component_association {, record_component_association}
 | null record
```

4/2
```
record_component_association ::=
   [component_choice_list =>] expression
 | component_choice_list => <>
```

5/5
```
component_choice_list ::=
    component_selector_name {'|' component_selector_name}
 | others
```

A record_component_association is a *named component association* if it has a component_choice_list; otherwise, it is a *positional component association*. Any positional component associations shall precede any named component associations. If there is a named association with a component_choice_list of **others**, it shall come last. 6

In the record_component_association_list for a record_aggregate, if there is only one association, it shall be a named association. 7

Name Resolution Rules

The expected type for a record_aggregate shall be a single record type or record extension. 8/2

For the record_component_association_list of a record_aggregate, all components of the composite value defined by the aggregate are *needed*; for the association list of an extension_aggregate, only those components not determined by the ancestor expression or subtype are needed (see 4.3.2). Each *component_*selector_name in a record_component_association of a record_aggregate or extension_aggregate shall denote a needed component (including possibly a discriminant). Each *component_*selector_name in a record_component_association of a record_delta_aggregate (see 4.3.4) shall denote a nondiscriminant component of the type of the aggregate. 9/5

The expected type for the expression of a record_component_association is the type of the *associated* component(s); the associated component(s) are as follows: 10

- For a positional association, the component (including possibly a discriminant) in the corresponding relative position (in the declarative region of the type), counting only the needed components; 11
- For a named association with one or more *component_*selector_names, the named component(s); 12
- For a named association with the reserved word **others**, all needed components that are not associated with some previous association. 13

Legality Rules

If the type of a record_aggregate is a record extension, then it shall be a descendant of a record type, through one or more record extensions (and no private extensions). 14

A record_component_association_list shall be **null record** only if the list occurs in a record_aggregate or extension_aggregate, and there are no components needed for that list. 15/5

For a record_aggregate or extension_aggregate, each record_component_association other than an **others** choice with a <> shall have at least one associated component, and each needed component shall be associated with exactly one record_component_association. For a record_delta_aggregate, each *component_*selector_name of each component_choice_list shall denote a distinct nondiscriminant component of the type of the aggregate. 16/5

If a record_component_association with an expression has two or more associated components, all of them shall be of the same type, or all of them shall be of anonymous access types whose subtypes statically match. In addition, Legality Rules are enforced separately for each associated component. 16.1/5

For a record_aggregate or extension_aggregate, if a variant_part *P* is nested within a variant *V* that is not selected by the discriminant value governing the variant_part enclosing *V*, then there is no restriction on the discriminant governing *P*. Otherwise, the value of the discriminant that governs *P* shall be given by a static expression, or by a nonstatic expression having a constrained static nominal subtype. There shall be exactly one discrete_choice_list of *P* that covers every possible value of the expression, which in the 17/6

case of a nonstatic expression includes each value that belongs to the nominal subtype and satisfies the predicates of the subtype, and there shall be at least one such value.

17.1/2 A record_component_association for a discriminant without a default_expression shall have an expression rather than <>.

17.2/5 A record_component_association of the record_component_association_list of a record_delta_aggregate shall not:

17.3/5 • use the box compound delimiter <> rather than an expression;

17.4/5 • have an expression of a limited type;

17.5/5 • omit the component_choice_list; or

17.6/5 • have a component_choice_list that is an **others** choice.

17.7/5 For a record_delta_aggregate, no two *component*_selector_names shall denote components declared within different variants of the same variant_part.

Dynamic Semantics

18 The evaluation of a record_aggregate consists of the evaluation of the record_component_association_list.

19/5 For the evaluation of a record_component_association_list, any per-object constraints (see 3.8) for components specified in the association list are elaborated and any expressions are evaluated and converted to the subtype of the associated component. Any constraint elaborations and expression evaluations (and conversions) occur in an arbitrary order, except that the expression for a discriminant is evaluated (and converted) prior to the elaboration of any per-object constraint that depends on it, which in turn occurs prior to the evaluation and conversion of the expression for the component with the per-object constraint. If the value of a discriminant that governs a selected variant_part is given by a nonstatic expression, and the evaluation of that expression yields a value that does not belong to the nominal subtype of the expression, then Constraint_Error is raised.

19.1/2 For a record_component_association with an expression, the expression defines the value for the associated component(s). For a record_component_association with <>, if the component_declaration has a default_expression, that default_expression defines the value for the associated component(s); otherwise, the associated component(s) are initialized by default as for a stand-alone object of the component subtype (see 3.3.1).

20 The expression of a record_component_association is evaluated (and converted) once for each associated component.

21/6 The nominal subtype of a record_aggregate is constrained by the values defined for the discriminants, if any.

22/5 NOTE By the rules given above, for a record_aggregate with positional associations, expressions specifying discriminant values appear first and in the same order as their corresponding discriminant_specifications, since the known_discriminant_part occurs first in the declaration of the type.

Examples

23 *Example of a record aggregate with positional associations:*

24
```
(4, July, 1776)                                    -- see 3.8
```

Examples of record aggregates with named associations:

```
(Day => 4, Month => July, Year => 1776)
(Month => July, Day => 4, Year => 1776)

(Disk, Closed, Track => 5, Cylinder => 12)          -- see 3.8.1
(Unit => Disk, Status => Closed, Cylinder => 9, Track => 1)
```

Examples of component associations with several choices:

```
(Value => 0, Succ|Pred => new Cell'(0, null, null))  -- see 3.10.1
   -- The allocator is evaluated twice:
   -- Succ and Pred designate different cells

(Value => 0, Succ|Pred => <>)                        -- see 3.10.1
   -- Succ and Pred will be set to null
```

Examples of record aggregates for tagged types (see 3.9 and 3.9.1):

```
Expression'(null record)
Literal'(Value => 0.0)
Painted_Point'(0.0, Pi/2.0, Paint => Red)
```

4.3.2 Extension Aggregates

An extension_aggregate specifies a value for a type that is a record extension by specifying a value or subtype for an ancestor of the type, followed by associations for any components not determined by the ancestor_part. 1

Syntax

extension_aggregate ::= 2
  (ancestor_part **with** record_component_association_list)

ancestor_part ::= expression | subtype_mark 3

Name Resolution Rules

The expected type for an extension_aggregate shall be a single type that is a record extension. If the ancestor_part is an expression, it is expected to be of any tagged type. 4/2

Legality Rules

If the ancestor_part is a subtype_mark, it shall denote a specific tagged subtype. If the ancestor_part is an expression, it shall not be dynamically tagged. The type of the extension_aggregate shall be a descendant of the type of the ancestor_part (the *ancestor* type), through one or more record extensions (and no private extensions). If the ancestor_part is a subtype_mark, the view of the ancestor type from which the type is descended (see 7.3.1) shall not have unknown discriminants. 5/3

If the type of the ancestor_part is limited and at least one component is needed in the record_component_association_list, then the ancestor_part shall not have an operative constituent expression (see 4.4) that is a call to a function with an unconstrained result subtype. 5.1/5

Static Semantics

For the record_component_association_list of an extension_aggregate, the only components *needed* are those of the composite value defined by the aggregate that are not inherited from the type of the ancestor_part, plus any inherited discriminants if the ancestor_part is a subtype_mark that denotes an unconstrained subtype. 6

Dynamic Semantics

7 For the evaluation of an extension_aggregate, the record_component_association_list is evaluated. If the ancestor_part is an expression, it is also evaluated; if the ancestor_part is a subtype_mark, the components of the value of the aggregate not given by the record_component_association_list are initialized by default as for an object of the ancestor type. Any implicit initializations or evaluations are performed in an arbitrary order, except that the expression for a discriminant is evaluated prior to any other evaluation or initialization that depends on it.

8/3 If the type of the ancestor_part has discriminants and the ancestor_part is not a subtype_mark that denotes an unconstrained subtype, then a check is made that each discriminant determined by the ancestor_part has the value specified for a corresponding discriminant, if any, either in the record_component_association_list, or in the derived_type_definition for some ancestor of the type of the extension_aggregate. Constraint_Error is raised if this check fails.

9/6 The nominal subtype of an extension_aggregate is constrained by the values defined for the discriminants, if any, of the object defined by the aggregate.

10/5 NOTE 1 By the rules given in 4.3.1, if all components of the value of the extension_aggregate are determined by the ancestor_part, then the record_component_association_list will be simply **null record**.

11 NOTE 2 If the ancestor_part is a subtype_mark, then its type can be abstract. If its type is controlled, then as the last step of evaluating the aggregate, the Initialize procedure of the ancestor type is called, unless the Initialize procedure is abstract (see 7.6).

Examples

12 *Examples of extension aggregates (for types defined in 3.9.1):*

13
```
Painted_Point'(Point with Red)
(Point'(P) with Paint => Black)
```

14/5
```
(Expression with Left =>  new Literal'(Value => 1.2),
                 Right => new Literal'(Value => 3.4))
Addition'(Binop with null record)
             -- presuming Binop is of type Binary_Operation
```

4.3.3 Array Aggregates

1 In an array_aggregate, a value is specified for each component of an array, either positionally or by its index. For a positional_array_aggregate, the components are given in increasing-index order, with a final **others**, if any, representing any remaining components. For a named_array_aggregate, the components are identified by the values covered by the discrete_choices.

Syntax

2/5
```
array_aggregate ::=
   positional_array_aggregate | null_array_aggregate | named_array_aggregate
```

3/5
```
positional_array_aggregate ::=
   (expression, expression {, expression})
  | (expression {, expression}, others => expression)
  | (expression {, expression}, others => <>)
  | '[' expression {, expression}[, others => expression] ']'
  | '[' expression {, expression}, others => <> ']'
```

3.1/5
```
null_array_aggregate ::= '[' ']'
```

named_array_aggregate ::= 4/5
   (array_component_association_list)
 | '[' array_component_association_list ']'

array_component_association_list ::= 4.1/5
   array_component_association {, array_component_association}

array_component_association ::= 5/5
   discrete_choice_list => expression
 | discrete_choice_list => <>
 | iterated_component_association

iterated_component_association ::= 5.1/5
   for defining_identifier **in** discrete_choice_list => expression
 | **for** iterator_specification => expression

An *n-dimensional* array_aggregate is one that is written as n levels of nested array_aggregates (or at the bottom level, equivalent string_literals). For the multidimensional case (n >= 2) the array_aggregates (or equivalent string_literals) at the n–1 lower levels are called *subaggregate*s of the enclosing n-dimensional array_aggregate. The expressions of the bottom level subaggregates (or of the array_aggregate itself if one-dimensional) are called the *array component expressions* of the enclosing n-dimensional array_aggregate. 6

The defining_identifier of an iterated_component_association declares an *index parameter*, an object of the corresponding index type. 6.1/5

Name Resolution Rules

The expected type for an array_aggregate (that is not a subaggregate) shall be a single array type. The component type of this array type is the expected type for each array component expression of the array_aggregate. 7/2

The expected type for each discrete_choice in any discrete_choice_list of a named_array_aggregate is the type of the *corresponding index*; the corresponding index for an array_aggregate that is not a subaggregate is the first index of its type; for an (n–m)-dimensional subaggregate within an array_aggregate of an n-dimensional type, the corresponding index is the index in position m+1. 8

Legality Rules

An array_aggregate of an n-dimensional array type shall be written as an n-dimensional array_aggregate, or as a null_array_aggregate. 9/5

An **others** choice is allowed for an array_aggregate only if an *applicable index constraint* applies to the array_aggregate. An applicable index constraint is a constraint provided by certain contexts that can be used to determine the bounds of the array value specified by an array_aggregate. Each of the following contexts (and none other) defines an applicable index constraint: 10/5

- For an explicit_actual_parameter, an explicit_generic_actual_parameter, the expression of a return statement, the return expression of an expression function, the initialization expression in an object_declaration, or a default_expression (for a parameter or a component), when the nominal subtype of the corresponding formal parameter, generic formal parameter, function return object, expression function return object, object, or component is a constrained array subtype, the applicable index constraint is the constraint of the subtype; 11/4
- For the expression of an assignment_statement where the name denotes an array variable, the applicable index constraint is the constraint of the array variable; 12

13 • For the operand of a qualified_expression whose subtype_mark denotes a constrained array subtype, the applicable index constraint is the constraint of the subtype;

14 • For a component expression in an aggregate, if the component's nominal subtype is a constrained array subtype, the applicable index constraint is the constraint of the subtype;

14.1/6 • For the *base*_expression of a delta_aggregate, the applicable index constraint is that of the delta_aggregate as a whole, if any;

15/3 • For a parenthesized expression, the applicable index constraint is that, if any, defined for the expression;

15.1/5 • For a conditional_expression (see 4.5.7), the applicable index constraint for each *dependent*_expression is that, if any, defined for the conditional_expression;

15.2/5 • For a declare_expression (see 4.5.9), the applicable index constraint for the *body*_expression is that, if any, defined for the declare_expression.

16 The applicable index constraint *applies* to an array_aggregate that appears in such a context, as well as to any subaggregates thereof. In the case of an explicit_actual_parameter (or default_expression) for a call on a generic formal subprogram, no applicable index constraint is defined.

17/6 The discrete_choice_list of an array_component_association (including an iterated_component_association) is allowed to have a discrete_choice that is a nonstatic choice_expression or that is a subtype_indication or range that defines a nonstatic or null range, only if it is the single discrete_choice of its discrete_choice_list, and either there is only one array_component_association in the enclosing array_component_association_list, not counting any component association with an **others** choice, or the enclosing aggregate is an array_delta_aggregate, not an array_aggregate.

17.1/6 Either all or none of the array_component_associations of an array_component_association_list shall be iterated_component_associations with an iterator_specification. An iterated_component_association with an iterator_specification shall not appear as the iterator_specification of an array_component_association of a multidimensional array_aggregate.

18/3 In a named_array_aggregate where all discrete_choices are static, no two discrete_choices are allowed to cover the same value (see 3.8.1); if there is no **others** choice, the discrete_choices taken together shall exactly cover a contiguous sequence of values of the corresponding index type.

19 A bottom level subaggregate of a multidimensional array_aggregate of a given array type is allowed to be a string_literal only if the component type of the array type is a character type; each character of such a string_literal shall correspond to a defining_character_literal of the component type.

19.1/6 An index parameter of an iterated_component_association of an aggregate shall not appear in the discrete_choice_list of an array_component_association of any subaggregate of that aggregate.

Static Semantics

20 A subaggregate that is a string_literal is equivalent to one that is a positional_array_aggregate of the same length, with each expression being the character_literal for the corresponding character of the string_literal.

20.1/5 The subtype (and nominal subtype) of an index parameter is the corresponding index subtype.

Dynamic Semantics

20.2/6 For an array_aggregate that contains only array_component_associations that are iterated_component_associations with iterator_specifications, evaluation proceeds in three steps:

1. Each iterator_specification is elaborated (in an arbitrary order); 20.3/6

2. An iteration is performed for each of these iterator_specifications solely to determine a count for the number of values conditionally produced by the iteration (see 5.5 and 5.5.2); all of these counts are combined to determine the overall length of the array, and ultimately the bounds of the array (defined below); 20.4/6

3. A second iteration is performed for each of the iterator_specifications, in the order given in the aggregate, and for each value conditionally produced by the iteration, the associated expression is evaluated, its value is converted to the component subtype of the array type, and used to define the value of the next component of the array starting at the low bound and proceeding sequentially toward the high bound. As part of this second iteration, a check is made that it results in the same number of elements as the first iteration; Constraint_Error is raised if this check fails. This check is performed before any attempt to access any nonexistent element of the array object. 20.5/6

The evaluation of any other array_aggregate of a given array type proceeds in two steps: 21/5

1. Any discrete_choices of this aggregate and of its subaggregates are evaluated in an arbitrary order, and converted to the corresponding index type; 22

2. The array component expressions of the aggregate are evaluated in an arbitrary order and their values are converted to the component subtype of the array type; an array component expression is evaluated once for each associated component. 23

Each expression in an array_component_association defines the value for the associated component(s). For an array_component_association with <>, the associated component(s) are initialized to the Default_Component_Value of the array type if this aspect has been specified for the array type; otherwise, they are initialized by default as for a stand-alone object of the component subtype (see 3.3.1). 23.1/4

During an evaluation of the expression of an iterated_component_association with a discrete_choice_list, the value of the corresponding index parameter is that of the corresponding index of the corresponding array component. During an evaluation of the expression of an iterated_component_association with an iterator_specification, the value of the loop parameter of the iterator_specification is the value produced by the iteration (as described in 5.5.2). 23.2/5

The bounds of the index range of an array_aggregate (including a subaggregate) are determined as follows: 24

- For an array_aggregate with an **others** choice, the bounds are those of the corresponding index range from the applicable index constraint; 25

- For a positional_array_aggregate (or equivalent string_literal) without an **others** choice, the lower bound is that of the corresponding index range in the applicable index constraint, if defined, or that of the corresponding index subtype, if not; in either case, the upper bound is calculated (see 3.6.1) from the lower bound and the number of expressions (or the length of the string_literal); 26/6

- For a null_array_aggregate, bounds for each dimension are determined as for a positional_array_aggregate without an **others** choice that has no expressions for each dimension; 26.1/5

- For a named_array_aggregate containing only iterated_component_associations with an iterator_specification, the lower bound is determined as for a positional_array_aggregate without an **others** choice, and the upper bound is calculated (see 3.6.1) from the lower bound and the total number of values produced by the set of iterations; 26.2/6

- For any other named_array_aggregate without an **others** choice, the bounds are determined by the smallest and largest index values covered by any discrete_choice_list. 27/5

28 For an array_aggregate, a check is made that the index range defined by its bounds is compatible with the corresponding index subtype.

29/3 For an array_aggregate with an **others** choice, a check is made that no expression or <> is specified for an index value outside the bounds determined by the applicable index constraint.

30 For a multidimensional array_aggregate, a check is made that all subaggregates that correspond to the same index have the same bounds.

31 The exception Constraint_Error is raised if any of the above checks fail.

31.1/6 The nominal subtype of an array_aggregate (other than a subaggregate) is constrained by the values defined for its bounds and those of any subaggregates.

Implementation Permissions

32/6 When evaluating iterated_component_associations for an array_aggregate that contains only iterated_component_associations with iterator_specifications, the second step of evaluating an iterated_component_association may be omitted if the implementation can determine the maximum number of values by some other means. Such "other means" can include making use of an applicable index constraint or the Length function of a suitable container type.

33/5 NOTE 1 In an array_aggregate delimited by parentheses, positional notation can only be used with two or more expressions; a single expression in parentheses is interpreted as a parenthesized expression. An array_aggregate delimited by square brackets can be used to specify an array with a single component.

34/5 NOTE 2 An index parameter is a constant object (see 3.3).

Examples

35 *Examples of array aggregates with positional associations:*

36
```
(7, 9, 5, 1, 3, 2, 4, 8, 6, 0)
Table'(5, 8, 4, 1, others => 0)   -- see 3.6
```

37 *Examples of array aggregates with named associations:*

38/5
```
(1 .. 5 => (1 .. 8 => 0.0))       -- two-dimensional
[1 .. N => new Cell]              -- N new cells, in particular for N = 0
```

39/5
```
Table'(2 | 4 | 10 => 1, others => 0)
Schedule'(Mon .. Fri => True,  others => False)   -- see 3.6
Schedule'[Wed | Sun  => False, others => True]
Vector'(1 => 2.5)                                 -- single-component vector
```

40 *Examples of two-dimensional array aggregates:*

41
```
-- Three aggregates for the same value of subtype Matrix(1..2,1..3) (see 3.6):
```

42/5
```
((1.1, 1.2, 1.3), (2.1, 2.2, 2.3))
(1 => [1.1, 1.2, 1.3], 2 => [2.1, 2.2, 2.3])
[1 => (1 => 1.1, 2 => 1.2, 3 => 1.3),
 2 => (1 => 2.1, 2 => 2.2, 3 => 2.3)]
```

43 *Examples of aggregates as initial values:*

44
```
A : Table := (7, 9, 5, 1, 3, 2, 4, 8, 6, 0);          -- A(1)=7, A(10)=0
B : Table := (2 | 4 | 10 => 1, others => 0);          -- B(1)=0, B(10)=1
C : constant Matrix := (1 .. 5 => (1 .. 8 => 0.0));
                                                      -- C'Last(1)=5, C'Last(2)=8
```

45
```
D : Bit_Vector(M .. N) := (M .. N => True);           -- see 3.6
E : Bit_Vector(M .. N) := (others => True);
F : String(1 .. 1) := (1 => 'F');  -- a one component aggregate: same as "F"
```

```
G : constant Matrix :=                                                    46/5
    (for I in 1 .. 4 =>
       (for J in 1 .. 4 =>
          (if I=J then 1.0 else 0.0)));  -- Identity matrix
Empty_Matrix : constant Matrix := [];  -- A matrix without elements       47/5
```

Example of an array aggregate with defaulted others choice and with an applicable index constraint provided by an enclosing record aggregate: 48/2

```
Buffer'(Size => 50, Pos => 1, Value => ('x', others => <>))  -- see 3.7   49/5
```

4.3.4 Delta Aggregates

Evaluating a (record or array) delta aggregate yields a composite value that starts with a copy of another value of the same type and then assigns to some (but typically not all) components of the copy. 1/5

Syntax

delta_aggregate ::= record_delta_aggregate | array_delta_aggregate 2/5

record_delta_aggregate ::= 3/5
(*base*_expression **with delta** record_component_association_list)

array_delta_aggregate ::= 4/5
(*base*_expression **with delta** array_component_association_list)
| '[' *base*_expression **with delta** array_component_association_list ']'

Name Resolution Rules

The expected type for a record_delta_aggregate shall be a single descendant of a record type or record extension. 5/5

The expected type for an array_delta_aggregate shall be a single array type. 6/5

The expected type for the *base*_expression of any delta_aggregate is the type of the enclosing delta_aggregate. 7/5

The Name Resolution Rules and Legality Rules for each record_component_association of a record_delta_aggregate are as defined in 4.3.1. 8/5

For an array_delta_aggregate, the expected type for each discrete_choice in an array_component_association is the index type of the type of the delta_aggregate. 9/5

The expected type of the expression in an array_component_association is defined as for an array_component_association occurring within an array_aggregate of the type of the delta_aggregate. 10/5

Legality Rules

For an array_delta_aggregate, the array_component_association shall not use the box symbol <>, and the discrete_choice shall not be **others**. 11/5

For an array_delta_aggregate, the dimensionality of the type of the delta_aggregate shall be 1. 12/5

For an array_delta_aggregate, the *base*_expression and each expression in every array_component_association shall be of a nonlimited type. 13/5

Dynamic Semantics

14/5 The evaluation of a delta_aggregate begins with the evaluation of the *base*_expression of the delta_aggregate; then that value is used to create and initialize the anonymous object of the aggregate. The bounds of the anonymous object of an array_delta_aggregate and the discriminants (if any) of the anonymous object of a record_delta_aggregate are those of the *base*_expression. If a record_delta_aggregate is of a specific tagged type, its tag is that of the specific type; if it is of a class-wide type, its tag is that of the *base*_expression.

15/5 For a record_delta_aggregate, for each component associated with each record_component_association (in an unspecified order):

16/5 - if the associated component belongs to a variant, a check is made that the values of the discriminants are such that the anonymous object has this component. The exception Constraint_Error is raised if this check fails.

17/5 - the expression of the record_component_association is evaluated, converted to the nominal subtype of the associated component, and assigned to the component of the anonymous object.

18/5 For an array_delta_aggregate, for each discrete_choice of each array_component_association (in the order given in the enclosing discrete_choice_list and array_component_association_list, respectively) the discrete_choice is evaluated; for each represented index value (in ascending order, if the discrete_choice represents a range):

19/5 - the index value is converted to the index type of the array type.

20/5 - a check is made that the index value belongs to the index range of the anonymous object of the aggregate; Constraint_Error is raised if this check fails.

21/5 - the component expression is evaluated, converted to the array component subtype, and assigned to the component of the anonymous object identified by the index value.

22/6 The nominal subtype of a delta_aggregate is that of its *base*_expression.

Examples

23/5 *Examples of use of delta aggregates in a postcondition:*

24/5
```
procedure Twelfth (D : in out Date) -- see 3.8 for type Date
   with Post => D = (D'Old with delta Day => 12);
```

25/5
```
procedure The_Answer (V : in out Vector;
                      A, B : in Integer) -- see 3.6 for type Vector
   with Post => V = (V'Old with delta A .. B => 42.0, V'First => 0.0);
```

26/5 *Examples where the base expression is nontrivial:*

27/5
```
New_Cell : Cell := (Min_Cell (Head) with delta Value => 42);
   -- see 3.10.1 for Cell and Head; 6.1 for Min_Cell
```

28/5
```
A1 : Vector := ((0 => 1.0, 1 => 2.0, 2 => 3.0)
       with delta Integer(Random * 2.0) => 14.2);
   -- see 3.6 for declaration of type Vector
   -- see 6.1 for declaration of Random
```

29/5
```
Tomorrow := ((Yesterday with delta Day => 12)
                    with delta Month => April); -- see 3.8
```

30/5 *Example where the base expression is class-wide:*

31/5
```
function Translate (P : Point'Class; X, Y : Real) return Point'Class is
   (P with delta X => P.X + X,
               Y => P.Y + Y); -- see 3.9 for declaration of type Point
```

4.3.5 Container Aggregates

In a container_aggregate, values are specified for elements of a container; for a positional_container_aggregate, the elements are given sequentially; for a named_container_aggregate, the elements are specified by a sequence of key/value pairs, or using an iterator. The Aggregate aspect of the type of the aggregate determines how the elements are combined to form the container. 1/5

For a type other than an array type, the following type-related operational aspect may be specified: 2/5

Aggregate This aspect is an aggregate of the form: 3/5

(Empty => name[, 4/5
Add_Named => *procedure*_name][,
Add_Unnamed => *procedure*_name][,
New_Indexed => *function*_name,
Assign_Indexed => *procedure*_name])

The type for which this aspect is specified is known as the *container type* of the Aggregate aspect. A *procedure*_name shall be specified for at least one of Add_Named, Add_Unnamed, or Assign_Indexed. If Add_Named is specified, neither Add_Unnamed nor Assign_Indexed shall be specified. Either both or neither of New_Indexed and Assign_Indexed shall be specified. 5/5

Name Resolution Rules

The name specified for Empty for an Aggregate aspect shall denote exactly one function with a result type of the container type that has no parameters, or that has one **in** parameter of a signed integer type. 6/6

The *procedure*_name specified for Add_Unnamed for an Aggregate aspect shall denote exactly one procedure that has two parameters, the first an **in out** parameter of the container type, and the second an **in** parameter whose type is called the *element type* of the container type. 7/6

The *function*_name specified for New_Indexed for an Aggregate aspect shall denote exactly one function with a result type of the container type, and two parameters of the same type, called the *key type* of the container type. 8/6

The *procedure*_name specified for Add_Named or Assign_Indexed for an Aggregate aspect shall denote exactly one procedure that has three parameters, the first an **in out** parameter of the container type, the second an **in** parameter whose type is called the *key type* of the container type, and the third, an **in** parameter whose type is called the *element type* of the container type. 9/6

Legality Rules

If the container type of an Aggregate aspect is a private type, the full type of the container type shall not be an array type. 10/6

For an Aggregate aspect, neither the element type nor the key type (if any) of the container type shall be a limited type. Additionally, the key type of Assign_Indexed shall be the same type as that of the parameters of New_Indexed, and that type shall be a discrete type. If both Add_Unnamed and Assign_Indexed are specified, the final parameters shall be of the same type — the element type of the container type. 11/6

None of the subprograms specified for an Aggregate aspect shall have a formal access parameter, nor an explicitly aliased formal parameter. 11.1/6

If the container type *T* is not abstract, then none of the subprograms specified in the Aggregate aspect for *T* shall be abstract. 11.2/6

11.3/6 In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

Static Semantics

12/5 The Aggregate aspect is nonoverridable (see 13.1.1).

Syntax

13/5
```
container_aggregate ::=
   null_container_aggregate
 | positional_container_aggregate
 | named_container_aggregate
```

14/5 null_container_aggregate ::= '[' ']'

15/5 positional_container_aggregate ::= '[' expression{, expression} ']'

16/5 named_container_aggregate ::= '[' container_element_association_list ']'

17/5
```
container_element_association_list ::=
   container_element_association {, container_element_association}
```

18/5
```
container_element_association ::=
   key_choice_list => expression
 | key_choice_list => <>
 | iterated_element_association
```

19/5 key_choice_list ::= key_choice {'|' key_choice}

20/5 key_choice ::= *key*_expression | discrete_range

21/5
```
iterated_element_association ::=
   for loop_parameter_specification[ use key_expression] => expression
 | for iterator_specification[ use key_expression] => expression
```

Name Resolution Rules

22/6 The expected type for a container_aggregate shall be a single type for which the Aggregate aspect has been specified, or a class-wide type rooted at such a type. The expected type for each expression of a container_aggregate is the element type of the expected type. For the purposes of this subclause, the expected type *has an Aggregate aspect* if it is class-wide and the aspect has been specified for its root type.

23/5 The expected type for a *key*_expression, or a discrete_range of a key_choice, is the key type of the expected type of the aggregate.

Legality Rules

24/6 The expected type for a positional_container_aggregate shall have an Aggregate aspect that includes a specification for an Add_Unnamed procedure or an Assign_Indexed procedure. The expected type for a named_container_aggregate that contains one or more iterated_element_associations with a *key*_expression shall have an Aggregate aspect that includes a specification for the Add_Named procedure. The expected type for a named_container_aggregate that contains one or more key_choice_lists shall have an Aggregate aspect that includes a specification for the Add_Named or Assign_Indexed procedure. For other kinds of named_container_aggregates, there are no additional

restrictions on the Aggregate aspect of the expected type. A null_container_aggregate can be of any type with an Aggregate aspect.

A non-null container aggregate is called an *indexed aggregate* if the expected type *T* of the aggregate specifies an Assign_Indexed procedure in its Aggregate aspect, and either there is no Add_Unnamed procedure specified for the type, or the aggregate is a named_container_aggregate with a container_element_association that contains a key_choice_list or a loop_parameter_specification. The key type of an indexed aggregate is also called the *index type* of the aggregate. 25/5

A container_element_association with a <> rather than an expression is permitted only in an indexed aggregate. 26/6

For an iterated_element_association without a *key*_expression, if the aggregate is an indexed aggregate or the expected type of the aggregate specifies an Add_Named procedure in its Aggregate aspect, then the type of the loop parameter of the iterated_element_association shall be the same as the key type of the aggregate. 27/5

For a named_container_aggregate that is an indexed aggregate, all container_element_associations shall contain either a key_choice_list, or a loop_parameter_specification without a *key*_expression or iterator_filter. Furthermore, for such an aggregate, either: 28/5

- all key_choices shall be static expressions or static ranges, and every loop_parameter_specification shall have a discrete_subtype_definition that defines a non-null static range, and the set of values of the index type covered by the key_choices and the discrete_subtype_definitions shall form a contiguous range of values with no duplications; or 29/5
- there shall be exactly one container_element_association, and if it has a key_choice_list, the list shall have exactly one key_choice. 30/5

Dynamic Semantics

The evaluation of a container_aggregate starts by creating an anonymous object *A* of the expected type *T*, initialized as follows: 31/5

- if the aggregate is an indexed aggregate, from the result of a call on the New_Indexed function; the actual parameters in this call represent the lower and upper bound of the aggregate, and are determined as follows: 32/5
  - if the aggregate is a positional_container_aggregate, the lower bound is the low bound of the subtype of the key parameter of the Assign_Indexed procedure, and the upper bound has a position number that is the sum of the position number of the lower bound and one less than the number of expressions in the aggregate; 33/6
  - if the aggregate is a named_container_aggregate, the lower bound is the lowest value covered by a key_choice_list or is the low bound of a range defined by a discrete_subtype_definition of a loop_parameter_specification; the upper bound is the highest value covered by a key_choice_list or is the high bound of a range defined by a discrete_subtype_definition of a loop_parameter_specification. 34/5
- if the aggregate is not an indexed aggregate, from a call on the Empty function specified in the Aggregate aspect. In the case of an Empty function with a formal parameter, the actual parameter has the following value: 35/6
  - for a null_container_aggregate, the value zero; 36/5
  - for a positional_container_aggregate, the number of expressions; 37/5
  - *This paragraph was deleted.* 38/6

39/6 - for a named_container_aggregate where each iterated_element_association, if any, contains a loop_parameter_specification, the total number of elements specified by all of the container_element_associations;

40/5 - otherwise, to an implementation-defined value.

41/5 The evaluation then proceeds as follows:

42/5 - for a null_container_aggregate, the anonymous object *A* is the result;

43/5 - for a positional_container_aggregate of a type with a specified Add_Unnamed procedure, each expression is evaluated in an arbitrary order, and the Add_Unnamed procedure is invoked in sequence with the anonymous object *A* as the first parameter and the result of evaluating each expression as the second parameter, in the order of the expressions;

44/6 - for a positional_container_aggregate that is an indexed aggregate, each expression is evaluated in an arbitrary order, and the Assign_Indexed procedure is invoked once for each expression of the aggregate with the anonymous object *A* as the first parameter, the key value as the second parameter, computed by starting with the low bound of the subtype of the key formal parameter of the Assign_Indexed procedure and taking the successor of this value for each successive expression of the aggregate, and the result of evaluating each expression as the third parameter;

45/5 - for a named_container_aggregate for a type with an Add_Named procedure in its Aggregate aspect, the container_element_associations are evaluated in an arbitrary order:

46/6 - for a container_element_association with a key_choice_list, for each key_choice of the list in an arbitrary order, the key_choice is evaluated, and the Add_Named procedure is invoked once for each value covered by the key_choice (in an arbitrary order), with the anonymous object *A* as the first parameter, the value from the key_choice as the second parameter, and the result of evaluating the expression of the container_element_association as the third parameter;

47/5 - for a container_element_association with an iterated_element_association, first the iterated_element_association is elaborated, then an iteration is performed, and for each value conditionally produced by the iteration (see 5.5 and 5.5.2) the Add_Named procedure is invoked with the anonymous object *A* as the first parameter, the result of evaluating the expression as the third parameter, and:

48/5 - if there is a *key*_expression, the result of evaluating the *key*_expression as the second parameter;

49/5 - otherwise, with the loop parameter as the second parameter;

50/5 - for a named_container_aggregate that is an indexed aggregate, the evaluation proceeds as above for the case of Add_Named, but with the Assign_Indexed procedure being invoked instead of Add_Named; in the case of a container_element_association with a <> rather than an expression, the corresponding call on Assign_Indexed is not performed, leaving the component as it was upon return from the New_Indexed function;

51/5 - for any other named_container_aggregate, the container_element_associations (which are necessarily iterated_element_associations) are evaluated in the order given; each such evaluation comprises two steps:

52/5 1. the iterated_element_association is elaborated;

53/5 2. an iteration is performed, and for each value conditionally produced by the iteration (see 5.5 and 5.5.2) the Add_Unnamed procedure is invoked, with the anonymous object *A* as the first parameter and the result of evaluating the expression as the second parameter.

54/6 When the above wording says that a subprogram is invoked or called, this is a subprogram call as defined in 6.4, with parameter associations as specified in the wording evaluated as defined 6.4.1. In particular,

this means that the parameters are converted to the subtype of the formal parameter (which can raise an exception — see 4.6). Furthermore, if the expected type is class-wide, the semantics defined in 3.9.2 for tag-indeterminate expressions apply to the call on Empty or New_Indexed, and those for controlling tags apply to the (dispatching) calls on Add_Named, Add_Unnamed, and Assign_Indexed.

The nominal subtype for an indexed aggregate is the return subtype of the New_Indexed function. The nominal subtype for any other container aggregate is the return subtype of the Empty function. 55/6

Examples

Examples of specifying the Aggregate aspect for a Set_Type, a Map_Type, and a Vector_Type: 56/5

```
   -- Set_Type is a set-like container type.                                    57/5
   type Set_Type is private
      with Aggregate => (Empty       => Empty_Set,
                         Add_Unnamed => Include);
   function Empty_Set return Set_Type;

   subtype Small_Int is Integer range -1000..1000;                             58/5

   procedure Include (S : in out Set_Type; N : in Small_Int);                  59/5

   -- Map_Type is a map-like container type.                                    60/5
   type Map_Type is private
      with Aggregate =>  (Empty     => Empty_Map,
                          Add_Named => Add_To_Map);

   procedure Add_To_Map (M     : in out Map_Type;                              61/5
                         Key   : in Integer;
                         Value : in String);

   function Empty_Map return Map_Type;                                         62/6

   -- Vector_Type is an extensible array-like container type.                  63/5
   type Vector_Type is private
      with Aggregate => (Empty          => Empty_Vector,
                         Add_Unnamed    => Append_One,
                         New_Indexed    => New_Vector,
                         Assign_Indexed => Assign_Element);

   function Empty_Vector (Capacity : Natural := 0) return Vector_Type;         64/6

   procedure Append_One (V : in out Vector_Type; New_Item : in String);        65/5

   procedure Assign_Element (V     : in out Vector_Type;                       66/5
                             Index : in Positive;
                             Item  : in String);

   function New_Vector (First, Last : Positive) return Vector_Type             67/5
      with Pre => First = Positive'First;
      -- Vectors are always indexed starting at the
      -- lower bound of their index subtype.

-- Private part not shown.                                                     68/5
```

Examples of container aggregates for Set_Type, Map_Type, and Vector_Type: 69/5

```
-- Example aggregates using Set_Type.                                         70/5
S : Set_Type;

-- Assign the empty set to S:                                                 71/5
S := [];

-- Is equivalent to:                                                          72/5
S := Empty_Set;

-- A positional set aggregate:                                                73/5
S := [1, 2];

-- Is equivalent to:                                                          74/5
S := Empty_Set;
Include (S, 1);
Include (S, 2);
```

75/5
```
-- A set aggregate with an iterated_element_association:
S := [for Item in 1 .. 5 => Item * 2];
```

76/5
```
-- Is equivalent to:
S := Empty_Set;
for Item in 1 .. 5 loop
   Include (S, Item * 2);
end loop;
```

77/5
```
-- A set aggregate consisting of two iterated_element_associations:
S := [for Item in 1 .. 5 => Item,
      for Item in 1 .. 5 => -Item];
```

78/5
```
-- Is equivalent (assuming set semantics) to:
S := Empty_Set;
for Item in 1 .. 5 loop
   Include (S, Item);
end loop;
for Item in -5 .. -1 loop
   Include (S, Item);
end loop;
```

79/5
```
-- Example aggregates using Map_Type.
M : Map_Type;
```

80/5
```
-- A simple named map aggregate:
M := [12 => "house", 14 => "beige"];
```

81/5
```
-- Is equivalent to:
M := Empty_Map;
Add_To_Map (M, 12, "house");
Add_To_Map (M, 14, "beige");
```

82/5
```
-- Define a table of pairs:
type Pair is record
   Key : Integer;
   Value : access constant String;
end record;
```

83/5
```
Table : constant array(Positive range <>) of Pair :=
   [(Key => 33, Value => new String'("a nice string")),
    (Key => 44, Value => new String'("an even better string"))];
```

84/5
```
-- A map aggregate using an iterated_element_association
-- and a key_expression, built from from a table of key/value pairs:
M := [for P of Table use P.Key => P.Value.all];
```

85/5
```
-- Is equivalent to:
M := Empty_Map;
for P of Table loop
   Add_To_Map (M, P.Key, P.Value.all);
end loop;
```

86/5
```
-- Create an image table for an array of integers:
Keys : constant array(Positive range <>) of Integer := [2, 3, 5, 7, 11];
```

87/5
```
-- A map aggregate where the values produced by the
-- iterated_element_association are of the same type as the key
-- (hence a separate key_expression is unnecessary):
M := [for Key of Keys => Integer'Image (Key)];
```

88/5
```
-- Is equivalent to:
M := Empty_Map;
for Key of Keys loop
   Add_To_Map (M, Key, Integer'Image (Key));
end loop;
```

89/5
```
-- Example aggregates using Vector_Type.
V : Vector_Type;
```

90/5
```
-- A positional vector aggregate:
V := ["abc", "def"];
```

```
-- Is equivalent to:                                                    91/5
V := Empty_Vector (2);
Append_One (V, "abc");
Append_One (V, "def");
-- An indexed vector aggregate:                                         92/5
V := [1 => "this", 2 => "is", 3 => "a", 4 => "test"];
-- Is equivalent to:                                                    93/5
V := New_Vector (1, 4);
Assign_Element (V, 1, "this");
Assign_Element (V, 2, "is");
Assign_Element (V, 3, "a");
Assign_Element (V, 4, "test");
```

4.4 Expressions

An *expression* is a formula that defines the computation or retrieval of a value. In this Reference Manual, the term "expression" refers to a construct of the syntactic category expression or of any of the following categories: choice_expression, choice_relation, relation, simple_expression, term, factor, primary, conditional_expression, quantified_expression. 1/3

Syntax

expression ::= 2
    relation {**and** relation} | relation {**and then** relation}
  | relation {**or** relation} | relation {**or else** relation}
  | relation {**xor** relation}

choice_expression ::= 2.1/3
    choice_relation {**and** choice_relation}
  | choice_relation {**or** choice_relation}
  | choice_relation {**xor** choice_relation}
  | choice_relation {**and then** choice_relation}
  | choice_relation {**or else** choice_relation}

choice_relation ::= 2.2/3
    simple_expression [relational_operator simple_expression]

relation ::= 3/4
    simple_expression [relational_operator simple_expression]
  | *tested*_simple_expression [**not**] **in** membership_choice_list
  | raise_expression

membership_choice_list ::= membership_choice {'|' membership_choice} 3.1/5

membership_choice ::= *choice*_simple_expression | range | subtype_mark 3.2/4

simple_expression ::= 4
   [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor} 5

factor ::= primary [** primary] | **abs** primary | **not** primary 6

7/5 primary ::=
numeric_literal | **null** | string_literal | aggregate
| name | allocator | (expression)
| (conditional_expression) | (quantified_expression)
| (declare_expression)

Name Resolution Rules

8 A name used as a primary shall resolve to denote an object or a value.

8.1/6 A primary that is an expression surrounded by (and) is known as a *parenthesized expression*; the expression of a parenthesized expression is the *operand* of the parenthesized expression. If a parenthesized expression is expected to be of type *T*, the expected type of the operand is *T*. Similarly, if the parenthesized expression is expected to be of some class of types, the operand of the parenthesized expression is expected to be of the same class of types.

Static Semantics

9 Each expression has a type; it specifies the computation or retrieval of a value of that type.

9.1/6 *This paragraph was deleted.*

9.2/5 Every name or expression consists of one or more *operative constituent* names or expressions, only one of which is evaluated as part of evaluating the name or expression (the *evaluated operative constituent*). The operative constituents are determined as follows, according to the form of the expression (or name):

9.3/5 - if the expression is a conditional_expression, the operative constituents of its *dependent_*expressions;

9.4/5 - if the expression (or name) is a parenthesized expression, a qualified_expression, or a view conversion, the operative constituent(s) of its operand;

9.5/5 - if the expression is a declare_expression, the operative constituent(s) of its *body_*expression;

9.6/5 - otherwise, the expression (or name) itself.

9.7/5 In certain contexts, we specify that an operative constituent shall (or shall not) be *newly constructed*. This means the operative constituent shall (or shall not) be an aggregate or a function_call; in either case, a raise_expression is permitted.

9.8/6 An *iterative context* is a context for an expression that can be evaluated multiple times, and includes the following:

9.9/6 - the predicate of a quantified_expression;

9.10/6 - the expression or <> of an array_component_association;

9.11/6 - the expression or <> of a container_element_association;

9.12/6 - the *key_*expression or expression of an iterated_element_association.

Dynamic Semantics

10 The value of a primary that is a name denoting an object is the value of the object.

10.1/5 An expression of a numeric universal type is evaluated as if it has type *root_integer* (for *universal_integer*) or *root_real* (otherwise) unless the context identifies a specific type (in which case that type is used).

The evaluation of a parenthesized expression evaluates the operand of the parenthesized expression, and yields the result of the operand evaluation. 10.2/6

Implementation Permissions

For the evaluation of a primary that is a name denoting an object of an unconstrained numeric subtype, if the value of the object is outside the base range of its type, the implementation may either raise Constraint_Error or return the value of the object. 11

Examples

Examples of primaries: 12

```
4.0                 -- real literal                                          13
Pi                  -- named number
(1 .. 10 => 0)      -- array aggregate
Sum                 -- variable
Integer'Last        -- attribute
Sine(X)             -- function call
Color'(Blue)        -- qualified expression
Real(M*N)           -- conversion
(Line_Count + 10)   -- parenthesized expression
```

Examples of expressions: 14

```
Volume                          -- primary                                   15/2
not Destroyed                   -- factor
2*Line_Count                    -- term
-4.0                            -- simple expression
-4.0 + A                        -- simple expression
B**2 - 4.0*A*C                  -- simple expression
R*Sin(θ)*Cos(φ)                 -- simple expression
Password(1 .. 3) = "Bwv"        -- relation
Count in Small_Int              -- relation
Count not in Small_Int          -- relation
Index = 0 or Item_Hit           -- expression
(Cold and Sunny) or Warm        -- expression (parentheses are required)
A**(B**C)                       -- expression (parentheses are required)
```

4.5 Operators and Expression Evaluation

The language defines the following six categories of operators (given in order of increasing precedence). The corresponding operator_symbols, and only those, can be used as designators in declarations of functions for user-defined operators. See 6.6, “Overloading of Operators”. 1

Syntax

logical_operator ::= **and** | **or** | **xor** 2

relational_operator ::= = | /= | < | <= | > | >= 3

binary_adding_operator ::= + | – | & 4

unary_adding_operator ::= + | – 5

multiplying_operator ::= * | / | **mod** | **rem** 6

highest_precedence_operator ::= ** | **abs** | **not** 7

Static Semantics

8 For a sequence of operators of the same precedence level, the operators are associated with their operands in textual order from left to right. Parentheses can be used to impose specific associations.

9 For each form of type definition, certain of the above operators are *predefined*; that is, they are implicitly declared immediately after the type definition. For each such implicit operator declaration, the parameters are called Left and Right for *binary* operators; the single parameter is called Right for *unary* operators. An expression of the form X op Y, where op is a binary operator, is equivalent to a function_call of the form "op"(X, Y). An expression of the form op Y, where op is a unary operator, is equivalent to a function_call of the form "op"(Y). The predefined operators and their effects are described in subclauses 4.5.1 through 4.5.6.

Dynamic Semantics

10 The predefined operations on integer types either yield the mathematically correct result or raise the exception Constraint_Error. For implementations that support the Numerics Annex, the predefined operations on real types yield results whose accuracy is defined in Annex G, or raise the exception Constraint_Error.

Implementation Requirements

11 The implementation of a predefined operator that delivers a result of an integer or fixed point type may raise Constraint_Error only if the result is outside the base range of the result type.

12 The implementation of a predefined operator that delivers a result of a floating point type may raise Constraint_Error only if the result is outside the safe range of the result type.

Implementation Permissions

13 For a sequence of predefined operators of the same precedence level (and in the absence of parentheses imposing a specific association), an implementation may impose any association of the operators with operands so long as the result produced is an allowed result for the left-to-right association, but ignoring the potential for failure of language-defined checks in either the left-to-right or chosen order of association.

14 NOTE The two operands of an expression of the form X op Y, where op is a binary operator, are evaluated in an arbitrary order, as for any function_call (see 6.4).

Examples

15 *Examples of precedence:*

16
```
not Sunny or Warm      -- same as (not Sunny) or Warm
X > 4.0 and Y > 0.0    -- same as (X > 4.0) and (Y > 0.0)
```

17
```
-4.0*A**2              -- same as –(4.0 * (A**2))
abs(1 + A) + B         -- same as (abs (1 + A)) + B
Y**(-3)                -- parentheses are necessary
A / B * C              -- same as (A/B)*C
A + (B + C)            -- evaluate B + C before adding it to A
```

4.5.1 Logical Operators and Short-circuit Control Forms

Name Resolution Rules

1 An expression consisting of two relations connected by **and then** or **or else** (a *short-circuit control form*) shall resolve to be of some boolean type; the expected type for both relations is that same boolean type.

Static Semantics

The following logical operators are predefined for every boolean type *T*, for every modular type *T*, and for every one-dimensional array type *T* whose component type is a boolean type: 2

```
function "and"(Left, Right : T) return T
function "or" (Left, Right : T) return T
function "xor"(Left, Right : T) return T
```
3

For boolean types, the predefined logical operators **and**, **or**, and **xor** perform the conventional operations of conjunction, inclusive disjunction, and exclusive disjunction, respectively. 4

For modular types, the predefined logical operators are defined on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation for the result, where zero represents False and one represents True. If this result is outside the base range of the type, a final subtraction by the modulus is performed to bring the result into the base range of the type. 5

The logical operators on arrays are performed on a component-by-component basis on matching components (as for equality — see 4.5.2), using the predefined logical operator for the component type. The bounds of the resulting array are those of the left operand. 6

Dynamic Semantics

The short-circuit control forms **and then** and **or else** deliver the same result as the corresponding predefined **and** and **or** operators for boolean types, except that the left operand is always evaluated first, and the right operand is not evaluated if the value of the left operand determines the result. 7

For the logical operators on arrays, a check is made that for each component of the left operand there is a matching component of the right operand, and vice versa. Also, a check is made that each component of the result belongs to the component subtype. The exception Constraint_Error is raised if either of the above checks fails. 8

NOTE The conventional meaning of the logical operators is given by the following truth table: 9

A	B	(A **and** B)	(A **or** B)	(A **xor** B)
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

10

Examples

Examples of logical operators: 11

```
Sunny or Warm
Filter(1 .. 10) and Filter(15 .. 24)    -- see 3.6.1
```
12

Examples of short-circuit control forms: 13

```
Next_Car.Owner /= null and then Next_Car.Owner.Age > 25    -- see 3.10.1
N = 0 or else A(N) = Hit_Value
```
14

4.5.2 Relational Operators and Membership Tests

The *equality operators* = (equals) and /= (not equals) are predefined for nonlimited types. The other relational_operators are the *ordering operators* < (less than), <= (less than or equal), > (greater than), and >= (greater than or equal). The ordering operators are predefined for scalar types, and for *discrete array types*, that is, one-dimensional array types whose components are of a discrete type. 1

2/3 A *membership test*, using **in** or **not in**, determines whether or not a value belongs to any given subtype or range, is equal to any given value, has a tag that identifies a type that is covered by a given type, or is convertible to and has an accessibility level appropriate for a given access type. Membership tests are allowed for all types.

Name Resolution Rules

3/3 The *tested type* of a membership test is determined by the membership_choices of the membership_choice_list. Either all membership_choices of the membership_choice_list shall resolve to the same type, which is the tested type; or each membership_choice shall be of an elementary type, and the tested type shall be covered by each of these elementary types.

3.1/5 If the tested type is tagged, then the *tested*_simple_expression shall resolve to be of a type that is convertible (see 4.6) to the tested type; if untagged, the expected type of the *tested*_simple_expression is the tested type. The expected type of a *choice*_simple_expression in a membership_choice, and of a simple_expression of a range in a membership_choice, is the tested type of the membership operation.

Legality Rules

4/4 For a membership test, if the *tested*_simple_expression is of a tagged class-wide type, then the tested type shall be (visibly) tagged.

4.1/5 If a membership test includes one or more *choice*_simple_expressions and the tested type of the membership test is limited, then the tested type of the membership test shall have a visible primitive equality operator; if the tested type of the membership test is nonlimited with a user-defined primitive equality operator that is defined at a point where the type is limited, the tested type shall be a record type or record extension.

Static Semantics

5 The result type of a membership test is the predefined type Boolean.

6 The equality operators are predefined for every specific type *T* that is not limited, and not an anonymous access type, with the following specifications:

7
```
function "=" (Left, Right : T) return Boolean
function "/="(Left, Right : T) return Boolean
```

7.1/2 The following additional equality operators for the *universal_access* type are declared in package Standard for use with anonymous access types:

7.2/2
```
function "=" (Left, Right : universal_access) return Boolean
function "/="(Left, Right : universal_access) return Boolean
```

8 The ordering operators are predefined for every specific scalar type *T*, and for every discrete array type *T*, with the following specifications:

9
```
function "<" (Left, Right : T) return Boolean
function "<="(Left, Right : T) return Boolean
function ">" (Left, Right : T) return Boolean
function ">="(Left, Right : T) return Boolean
```

Name Resolution Rules

9.1/2 At least one of the operands of an equality operator for *universal_access* shall be of a specific anonymous access type. Unless the predefined equality operator is identified using an expanded name with prefix denoting the package Standard, neither operand shall be of an access-to-object type whose designated type is *D* or *D*'Class, where *D* has a user-defined primitive equality operator such that:

9.2/2 - its result type is Boolean;

- it is declared immediately within the same declaration list as *D* or any partial or incomplete view of *D*; and 9.3/3
- at least one of its operands is an access parameter with designated type *D*. 9.4/2

Legality Rules

At least one of the operands of the equality operators for *universal_access* shall be of type *universal_access*, or both shall be of access-to-object types, or both shall be of access-to-subprogram types. Further: 9.5/2

- When both are of access-to-object types, the designated types shall be the same or one shall cover the other, and if the designated types are elementary or array types, then the designated subtypes shall statically match; 9.6/2
- When both are of access-to-subprogram types, the designated profiles shall be subtype conformant. 9.7/2

If the profile of an explicitly declared primitive equality operator of an untagged record type is type conformant with that of the corresponding predefined equality operator, the declaration shall occur before the type is frozen. In addition, no type shall have been derived from the untagged record type before the declaration of the primitive equality operator. If the untagged record type is declared immediately within the visible part of a package, and the overriding primitive equality operator is explicitly declared within the private part of the package, the operator shall be subtype conformant with the predefined or inherited operator that it overrides. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 9.8/6

Dynamic Semantics

For discrete types, the predefined relational operators are defined in terms of corresponding mathematical operations on the position numbers of the values of the operands. 10

For real types, the predefined relational operators are defined in terms of the corresponding mathematical operations on the values of the operands, subject to the accuracy of the type. 11

Two access-to-object values are equal if they designate the same object, or if both are equal to the null value of the access type. 12

Two access-to-subprogram values are equal if they are the result of the same evaluation of an Access attribute_reference, or if both are equal to the null value of the access type. Two access-to-subprogram values are unequal if they designate different subprograms. It is unspecified whether two access values that designate the same subprogram but are the result of distinct evaluations of Access attribute_references are equal or unequal. 13

For an untagged record type, when outside the immediate scope of the type, any call on a primitive equals operator conformant with the profile of the predefined equals operator of the type, will invoke the body associated with the corresponding primitive equals operator visible at the end of the immediate scope of the type. 13.1/6

For a type extension, predefined equality is defined in terms of the primitive (possibly user-defined) equals operator for the parent type and for any components that have a record type in the extension part, and predefined equality for any other components not inherited from the parent type. 14/3

For a private type, if its full type is a record type or a record extension, predefined equality is defined in terms of the primitive equals operator of the full type; otherwise, predefined equality for the private type is that of its full type. 15/5

16 For other composite types, the predefined equality operators (and certain other predefined operations on composite types — see 4.5.1 and 4.6) are defined in terms of the corresponding operation on *matching components*, defined as follows:

17 • For two composite objects or values of the same non-array type, matching components are those that correspond to the same component_declaration or discriminant_specification;

18 • For two one-dimensional arrays of the same type, matching components are those (if any) whose index values match in the following sense: the lower bounds of the index ranges are defined to match, and the successors of matching indices are defined to match;

19 • For two multidimensional arrays of the same type, matching components are those whose index values match in successive index positions.

20 The analogous definitions apply if the types of the two objects or values are convertible, rather than being the same.

21 Given the above definition of matching components, the result of the predefined equals operator for composite types (other than for those composite types covered earlier) is defined as follows:

22 • If there are no components, the result is defined to be True;

23 • If there are unmatched components, the result is defined to be False;

24/3 • Otherwise, the result is defined in terms of the primitive equals operator for any matching components that are records, and the predefined equals for any other matching components.

24.1/5 If the primitive equals operator for an untagged record type is abstract, then Program_Error is raised at the point of any call to that abstract subprogram, implicitly as part of an equality operation on an enclosing composite object, or in an instance of a generic with a formal private type where the actual type is a record type with an abstract "=".

24.2/1 For any composite type, the order in which "=" is called for components is unspecified. Furthermore, if the result can be determined before calling "=" on some components, it is unspecified whether "=" is called on those components.

25 The predefined "/=" operator gives the complementary result to the predefined "=" operator.

26/3 For a discrete array type, the predefined ordering operators correspond to *lexicographic order* using the predefined order relation of the component type: A null array is lexicographically less than any array having at least one component. In the case of nonnull arrays, the left operand is lexicographically less than the right operand if the first component of the left operand is less than that of the right; otherwise, the left operand is lexicographically less than the right operand only if their first components are equal and the tail of the left operand is lexicographically less than that of the right (the *tail* consists of the remaining components beyond the first and can be null).

26.1/3 An *individual membership test* is the membership test of a single membership_choice.

27/4 For the evaluation of a membership test using **in** whose membership_choice_list has a single membership_choice, the *tested*_simple_expression and the membership_choice are evaluated in an arbitrary order; the result is the result of the individual membership test for the membership_choice.

27.1/4 For the evaluation of a membership test using **in** whose membership_choice_list has more than one membership_choice, the *tested*_simple_expression of the membership test is evaluated first and the result of the operation is equivalent to that of a sequence consisting of an individual membership test on each membership_choice combined with the short-circuit control form **or else**.

28/3 An individual membership test yields the result True if:

- The membership_choice is a *choice*_simple_expression, and the *tested*_simple_expression is equal to the value of the membership_choice. If the tested type is a record type or a record extension, or is limited at the point where the membership test occurs, the test uses the primitive equality for the type; otherwise, the test uses predefined equality. 28.1/5
- The membership_choice is a range and the value of the *tested*_simple_expression belongs to the given range. 28.2/4
- The membership_choice is a subtype_mark, the tested type is scalar, the value of the *tested*_simple_expression belongs to the range of the named subtype, and the value satisfies the predicates of the named subtype. 29/4
- The membership_choice is a subtype_mark, the tested type is not scalar, the value of the *tested*_simple_expression satisfies any constraints of the named subtype, the value satisfies the predicates of the named subtype, and: 30/4
  - if the type of the *tested*_simple_expression is class-wide, the value has a tag that identifies a type covered by the tested type; 30.1/4
  - if the tested type is an access type and the named subtype excludes null, the value of the *tested*_simple_expression is not null; 30.2/4
  - if the tested type is a general access-to-object type, the type of the *tested*_simple_expression is convertible to the tested type and its accessibility level is no deeper than that of the tested type; further, if the designated type of the tested type is tagged and the *tested*_simple_expression is nonnull, the tag of the object designated by the value of the *tested*_simple_expression is covered by the designated type of the tested type. 30.3/4

Otherwise, the test yields the result False. 31/3

A membership test using **not in** gives the complementary result to the corresponding membership test using **in**. 32

Implementation Requirements

For all nonlimited types declared in language-defined packages, the "=" and "/=" operators of the type shall behave as if they were the predefined equality operators for the purposes of the equality of composite types and generic formal types. 32.1/1

This paragraph was deleted. 33/2

NOTE If a composite type has components that depend on discriminants, two values of this type have matching components if and only if their discriminants are equal. Two nonnull arrays have matching components if and only if the length of each dimension is the same for both. 34

Examples

Examples of expressions involving relational operators and membership tests: 35

```
X /= Y                                                    36

A_String = "A"                          -- True (see 3.3.1)   37/5
"" < A_String and A_String < "Aa"       -- True
A_String < "Bb" and A_String < "A  "    -- True

My_Car = null          -- True if My_Car has been set to null (see 3.10.1)   38/3
My_Car = Your_Car              -- True if we both share the same car
My_Car.all = Your_Car.all      -- True if the two cars are identical

N not in 1 .. 10               -- range membership test   39/3
Today in Mon .. Fri            -- range membership test
Today in Weekday               -- subtype membership test (see 3.5.1)
Card in Clubs | Spades         -- list membership test (see 3.5.1)
Archive in Disk_Unit           -- subtype membership test (see 3.8.1)
Tree.all in Addition'Class     -- class membership test (see 3.9.1)
```

4.5.3 Binary Adding Operators

Static Semantics

1 The binary adding operators + (addition) and – (subtraction) are predefined for every specific numeric type *T* with their conventional meaning. They have the following specifications:

2
```
function "+"(Left, Right : T) return T
function "-"(Left, Right : T) return T
```

3 The concatenation operators & are predefined for every nonlimited, one-dimensional array type *T* with component type *C*. They have the following specifications:

4
```
function "&"(Left : T; Right : T) return T
function "&"(Left : T; Right : C) return T
function "&"(Left : C; Right : T) return T
function "&"(Left : C; Right : C) return T
```

Dynamic Semantics

5 For the evaluation of a concatenation with result type *T*, if both operands are of type *T*, the result of the concatenation is a one-dimensional array whose length is the sum of the lengths of its operands, and whose components comprise the components of the left operand followed by the components of the right operand. If the left operand is a null array, the result of the concatenation is the right operand. Otherwise, the lower bound of the result is determined as follows:

6 • If the ultimate ancestor of the array type was defined by a constrained_array_definition, then the lower bound of the result is that of the index subtype;

7 • If the ultimate ancestor of the array type was defined by an unconstrained_array_definition, then the lower bound of the result is that of the left operand.

8/6 The upper bound is calculated (see 3.6.1) from the lower bound and the length. A check is made that the upper bound of the result of the concatenation belongs to the range of the index subtype, unless the result is a null array. Constraint_Error is raised if this check fails.

9 If either operand is of the component type *C*, the result of the concatenation is given by the above rules, using in place of such an operand an array having this operand as its only component (converted to the component subtype) and having the lower bound of the index subtype of the array type as its lower bound.

10 The result of a concatenation is defined in terms of an assignment to an anonymous object, as for any function call (see 6.5).

11 NOTE As for all predefined operators on modular types, the binary adding operators + and – on modular types include a final reduction modulo the modulus if the result is outside the base range of the type.

Examples

12 *Examples of expressions involving binary adding operators:*

13
```
Z + 0.1       -- Z has to be of a real type
```

14
```
"A" & "BCD"  -- concatenation of two string literals
'A' & "BCD"  -- concatenation of a character literal and a string literal
'A' & 'A'    -- concatenation of two character literals
```

4.5.4 Unary Adding Operators

Static Semantics

The unary adding operators + (identity) and – (negation) are predefined for every specific numeric type *T* with their conventional meaning. They have the following specifications: 1

```
function "+"(Right : T) return T
function "-"(Right : T) return T
```
2

NOTE For modular integer types, the unary adding operator –, when given a nonzero operand, returns the result of subtracting the value of the operand from the modulus; for a zero operand, the result is zero. 3

4.5.5 Multiplying Operators

Static Semantics

The multiplying operators ***** (multiplication), / (division), **mod** (modulus), and **rem** (remainder) are predefined for every specific integer type *T*: 1

```
function "*"   (Left, Right : T) return T
function "/"   (Left, Right : T) return T
function "mod"(Left, Right : T) return T
function "rem"(Left, Right : T) return T
```
2

Signed integer multiplication has its conventional meaning. 3

Signed integer division and remainder are defined by the relation: 4

```
A = (A/B)*B + (A rem B)
```
5

where (A **rem** B) has the sign of A and an absolute value less than the absolute value of B. Signed integer division satisfies the identity: 6

```
(-A)/B = -(A/B) = A/(-B)
```
7

The signed integer modulus operator is defined such that the result of A **mod** B is either zero, or has the sign of B and an absolute value less than the absolute value of B; in addition, for some signed integer value N, this result satisfies the relation: 8/3

```
A = B*N + (A mod B)
```
9

The multiplying operators on modular types are defined in terms of the corresponding signed integer operators, followed by a reduction modulo the modulus if the result is outside the base range of the type (which is only possible for the "*" operator). 10

Multiplication and division operators are predefined for every specific floating point type *T*: 11

```
function "*"(Left, Right : T) return T
function "/"(Left, Right : T) return T
```
12

The following multiplication and division operators, with an operand of the predefined type Integer, are predefined for every specific fixed point type *T*: 13

```
function "*"(Left : T; Right : Integer) return T
function "*"(Left : Integer; Right : T) return T
function "/"(Left : T; Right : Integer) return T
```
14

All of the above multiplying operators are usable with an operand of an appropriate universal numeric type. The following additional multiplying operators for *root_real* are predefined, and are usable when both operands are of an appropriate universal or root numeric type, and the result is allowed to be of type *root_real*, as in a number_declaration: 15

16
```
function "*"(Left, Right : root_real) return root_real
function "/"(Left, Right : root_real) return root_real
```

17
```
function "*"(Left : root_real; Right : root_integer) return root_real
function "*"(Left : root_integer; Right : root_real) return root_real
function "/"(Left : root_real; Right : root_integer) return root_real
```

18 Multiplication and division between any two fixed point types are provided by the following two predefined operators:

19
```
function "*"(Left, Right : universal_fixed) return universal_fixed
function "/"(Left, Right : universal_fixed) return universal_fixed
```

Name Resolution Rules

19.1/2 The above two fixed-fixed multiplying operators shall not be used in a context where the expected type for the result is itself *universal_fixed* — the context has to identify some other numeric type to which the result is to be converted, either explicitly or implicitly. Unless the predefined universal operator is identified using an expanded name with prefix denoting the package Standard, an explicit conversion is required on the result when using the above fixed-fixed multiplication operator if either operand is of a type having a user-defined primitive multiplication operator such that:

19.2/3 - it is declared immediately within the same declaration list as the type or any partial or incomplete view thereof; and

19.3/2 - both of its formal parameters are of a fixed-point type.

19.4/2 A corresponding requirement applies to the universal fixed-fixed division operator.

Paragraph 20 was deleted.

Dynamic Semantics

21 The multiplication and division operators for real types have their conventional meaning. For floating point types, the accuracy of the result is determined by the precision of the result type. For decimal fixed point types, the result is truncated toward zero if the mathematical result is between two multiples of the *small* of the specific result type (possibly determined by context); for ordinary fixed point types, if the mathematical result is between two multiples of the *small*, it is unspecified which of the two is the result.

22 The exception Constraint_Error is raised by integer division, **rem**, and **mod** if the right operand is zero. Similarly, for a real type *T* with *T*'Machine_Overflows True, division by zero raises Constraint_Error.

23 NOTE 1 For positive A and B, A/B is the quotient and A **rem** B is the remainder when A is divided by B. The following relations are satisfied by the rem operator:

24
```
  A  rem (-B) =   A rem B
(-A) rem   B  = -(A rem B)
```

25 NOTE 2 For any signed integer K, the following identity holds:

26
```
A mod B   =   (A + K*B) mod B
```

The relations between signed integer division, remainder, and modulus are illustrated by the following table:

A	B	A/B	A **rem** B	A **mod** B	A	B	A/B	A **rem** B	A **mod** B
10	5	2	0	0	-10	5	-2	0	0
11	5	2	1	1	-11	5	-2	-1	4
12	5	2	2	2	-12	5	-2	-2	3
13	5	2	3	3	-13	5	-2	-3	2
14	5	2	4	4	-14	5	-2	-4	1

A	B	A/B	A **rem** B	A **mod** B	A	B	A/B	A **rem** B	A **mod** B
10	-5	-2	0	0	-10	-5	2	0	0
11	-5	-2	1	-4	-11	-5	2	-1	-1
12	-5	-2	2	-3	-12	-5	2	-2	-2
13	-5	-2	3	-2	-13	-5	2	-3	-3
14	-5	-2	4	-1	-14	-5	2	-4	-4

Examples

Examples of expressions involving multiplying operators:

```
I : Integer := 1;
J : Integer := 2;
K : Integer := 3;

X : Real := 1.0;                    -- see 3.5.7
Y : Real := 2.0;

F : Fraction := 0.25;               -- see 3.5.9
G : Fraction := 0.5;
```

Expression	*Value*	*Result Type*
I*J	2	*same as I and J, that is, Integer*
K/J	1	*same as K and J, that is, Integer*
K **mod** J	1	*same as K and J, that is, Integer*
X/Y	0.5	*same as X and Y, that is, Real*
F/2	0.125	*same as F, that is, Fraction*
3*F	0.75	*same as F, that is, Fraction*
0.75*G	0.375	*universal_fixed, implicitly convertible to any fixed point type*
Fraction(F*G)	0.125	*Fraction, as stated by the conversion*
Real(J)*Y	4.0	*Real, the type of both operands after conversion of J*

4.5.6 Highest Precedence Operators

Static Semantics

The highest precedence unary operator **abs** (absolute value) is predefined for every specific numeric type *T*, with the following specification:

```
function "abs"(Right : T) return T
```

The highest precedence unary operator **not** (logical negation) is predefined for every boolean type *T*, every modular type *T*, and for every one-dimensional array type *T* whose components are of a boolean type, with the following specification:

```
function "not"(Right : T) return T
```

The result of the operator **not** for a modular type is defined as the difference between the high bound of the base range of the type and the value of the operand. For a binary modulus, this corresponds to a bit-wise complement of the binary representation of the value of the operand.

The operator **not** that applies to a one-dimensional array of boolean components yields a one-dimensional boolean array with the same bounds; each component of the result is obtained by logical negation of the

corresponding component of the operand (that is, the component that has the same index value). A check is made that each component of the result belongs to the component subtype; the exception Constraint_Error is raised if this check fails.

7 The highest precedence *exponentiation* operator ** is predefined for every specific integer type *T* with the following specification:

8 `function "**"(Left : T; Right : Natural) return T`

9 Exponentiation is also predefined for every specific floating point type as well as *root_real*, with the following specification (where *T* is *root_real* or the floating point type):

10 `function "**"(Left : T; Right : Integer'Base) return T`

11/3 The right operand of an exponentiation is the *exponent*. The value of X**N with the value of the exponent N positive is the same as the value of X*X*...X (with N–1 multiplications) except that the multiplications are associated in an arbitrary order. With N equal to zero, the result is one. With the value of N negative (only defined for a floating point operand), the result is the reciprocal of the result using the absolute value of N as the exponent.

Implementation Permissions

12 The implementation of exponentiation for the case of a negative exponent is allowed to raise Constraint_Error if the intermediate result of the repeated multiplications is outside the safe range of the type, even though the final result (after taking the reciprocal) would not be. (The best machine approximation to the final result in this case would generally be 0.0.)

13 NOTE As implied by the specification given above for exponentiation of an integer type, a check is made that the exponent is not negative. Constraint_Error is raised if this check fails.

4.5.7 Conditional Expressions

1/3 A conditional_expression selects for evaluation at most one of the enclosed *dependent*_expressions, depending on a decision among the alternatives. One kind of conditional_expression is the if_expression, which selects for evaluation a *dependent*_expression depending on the value of one or more corresponding conditions. The other kind of conditional_expression is the case_expression, which selects for evaluation one of a number of alternative *dependent*_expressions; the chosen alternative is determined by the value of a *selecting*_expression.

Syntax

2/3 conditional_expression ::= if_expression | case_expression

3/3 if_expression ::=
  if condition **then** *dependent*_expression
  {**elsif** condition **then** *dependent*_expression}
  [**else** *dependent*_expression]

4/3 condition ::= *boolean*_expression

5/3 case_expression ::=
  case *selecting*_expression **is**
  case_expression_alternative {,
  case_expression_alternative}

case_expression_alternative ::= 6/3
  when discrete_choice_list =>
    *dependent*_expression

Wherever the Syntax Rules allow an expression, a conditional_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses. 7/3

Name Resolution Rules

If a conditional_expression is expected to be of a type *T*, then each *dependent*_expression of the conditional_expression is expected to be of type *T*. Similarly, if a conditional_expression is expected to be of some class of types, then each *dependent*_expression of the conditional_expression is subject to the same expectation. If a conditional_expression shall resolve to be of a type *T*, then each *dependent*_expression shall resolve to be of type *T*. 8/3

The possible types of a conditional_expression are further determined as follows: 9/3

- If the conditional_expression is the operand of a type conversion, the type of the conditional_expression is the target type of the conversion; otherwise, 10/3
- If all of the *dependent*_expressions are of the same type, the type of the conditional_expression is that type; otherwise, 11/3
- If a *dependent*_expression is of an elementary type, the type of the conditional_expression shall be covered by that type; otherwise, 12/3
- If the conditional_expression is expected to be of type *T* or shall resolve to type *T*, then the conditional_expression is of type *T*. 13/3

A condition is expected to be of any boolean type. 14/3

The expected type for the *selecting*_expression and the discrete_choices are as for case statements (see 5.4). 15/3

Legality Rules

All of the *dependent*_expressions shall be convertible (see 4.6) to the type of the conditional_expression. 16/3

If the expected type of a conditional_expression is a specific tagged type, all of the *dependent*_expressions of the conditional_expression shall be dynamically tagged, or none shall be dynamically tagged. In this case, the conditional_expression is dynamically tagged if all of the *dependent*_expressions are dynamically tagged, is tag-indeterminate if all of the *dependent*_expressions are tag-indeterminate, and is statically tagged otherwise. 17/3

If there is no **else** *dependent*_expression, the if_expression shall be of a boolean type. 18/3

All Legality Rules that apply to the discrete_choices of a case_statement (see 5.4) also apply to the discrete_choices of a case_expression except within an instance of a generic unit. 19/3

Dynamic Semantics

For the evaluation of an if_expression, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, the associated *dependent*_expression is evaluated, converted to the type of the if_expression, and the resulting value is the value of the if_expression. Otherwise (when there is no **else** clause), the value of the if_expression is True. 20/3

21/3 For the evaluation of a case_expression, the *selecting_*expression is first evaluated. If the value of the *selecting_*expression is covered by the discrete_choice_list of some case_expression_alternative, then the *dependent_*expression of the case_expression_alternative is evaluated, converted to the type of the case_expression, and the resulting value is the value of the case_expression. Otherwise (the value is not covered by any discrete_choice_list, perhaps due to being outside the base range), Constraint_Error is raised.

Examples

22/5 *Example of use of an* if_expression*:*

23/5
```
Put_Line ("Casey is " &
      (if Casey.Sex = M then "Male" else "Female")); -- see 3.10.1
```

24/5 *Example of use of a* case_expression*:*

25/5
```
function Card_Color (Card : Suit) return Color is -- see 3.5.1
  (case Card is
      when Clubs   | Spades   => Black,
      when Hearts  | Diamonds => Red);
```

4.5.8 Quantified Expressions

0.1/4 Quantified expressions provide a way to write universally and existentially quantified predicates over containers and arrays.

Syntax

1/3 quantified_expression ::=
  for quantifier loop_parameter_specification => predicate
 | **for** quantifier iterator_specification => predicate

2/3 quantifier ::= **all** | **some**

3/3 predicate ::= *boolean_*expression

4/3 Wherever the Syntax Rules allow an expression, a quantified_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses.

Name Resolution Rules

5/3 The expected type of a quantified_expression is any Boolean type. The predicate in a quantified_expression is expected to be of the same type.

Dynamic Semantics

6/5 For the evaluation of a quantified_expression, the loop_parameter_specification or iterator_specification is first elaborated. The evaluation of a quantified_expression then performs an iteration, and evaluates the predicate for each value conditionally produced by the iteration (see 5.5 and 5.5.2).

7/3 The value of the quantified_expression is determined as follows:

8/4 - If the quantifier is **all**, the expression is False if the evaluation of any predicate yields False; evaluation of the quantified_expression stops at that point. Otherwise (every predicate has been evaluated and yielded True), the expression is True. Any exception raised by evaluation of the predicate is propagated.

9/4 - If the quantifier is **some**, the expression is True if the evaluation of any predicate yields True; evaluation of the quantified_expression stops at that point. Otherwise (every predicate has been

evaluated and yielded False), the expression is False. Any exception raised by evaluation of the predicate is propagated.

Examples

Example of a quantified expression as a postcondition for a sorting routine on an array A with an index subtype T: 10/5

```
Post => (A'Length < 2 or else                                                    11/3
   (for all I in A'First .. T'Pred(A'Last) => A (I) <= A (T'Succ (I))))
```

Example of use of a quantified expression as an assertion that a positive number N is composite (as opposed to prime): 12/5

```
pragma Assert (for some X in 2 .. N when X * X <= N => N mod X = 0);             13/5
   -- see iterator_filter in 5.5
```

4.5.9 Declare Expressions

Declare expressions provide a way to declare local constants and object renamings in an expression context. 1/5

Syntax

declare_expression ::= 2/5
  declare {declare_item}
  begin *body*_expression

declare_item ::= object_declaration | object_renaming_declaration 3/5

Wherever the Syntax Rules allow an expression, a declare_expression may be used in place of the expression, so long as it is immediately surrounded by parentheses. 4/5

Legality Rules

A declare_item that is an object_declaration shall declare a constant of a nonlimited type. 5/5

A declare_item that is an object_renaming_declaration (see 8.5.1) shall not rename an object of a limited type if any operative constituent of the *object*_name is part of a value conversion or is part of a newly constructed object (see 4.4). 6/6

The following are not allowed within a declare_expression: a declaration containing the reserved word **aliased**; the attribute_designator Access or Unchecked_Access; an anonymous access type; or a pragma other than an executable pragma (see 2.8). 7/6

Name Resolution Rules

If a declare_expression is expected to be of a type *T*, then the *body*_expression is expected to be of type *T*. Similarly, if a declare_expression is expected to be of some class of types, then the *body*_expression is subject to the same expectation. If a declare_expression shall resolve to be of a type *T*, then the *body*_expression shall resolve to be of type *T*. 8/5

The type of a declare_expression is the type of the *body*_expression. 9/5

Dynamic Semantics

For the evaluation of a declare_expression, the declare_items are elaborated in order, and then the *body*_expression is evaluated. The value of the declare_expression is that of the *body*_expression. 10/5

Examples

11/5 *Example of use of a declare expression as a replacement postcondition for Ada.Containers.Vectors."&" (see A.18.2):*

12/5
```
with Post =>
   (declare
      Result renames Vectors."&"'Result;
      Length : constant Count_Type := Left.Length + Right.Length;
    begin
      Result.Length = Length and then
      not Tampering_With_Elements_Prohibited (Result) and then
      not Tampering_With_Cursors_Prohibited (Result) and then
      Result.Capacity >= Length)
```

4.5.10 Reduction Expressions

1/5 Reduction expressions provide a way to map or transform a collection of values into a new set of values, and then summarize the values produced by applying an operation to reduce the set to a single value result. A reduction expression is represented as an attribute_reference of the reduction attributes Reduce or Parallel_Reduce.

Syntax

2/5 reduction_attribute_reference ::=
  value_sequence'reduction_attribute_designator
 | prefix'reduction_attribute_designator

3/5 value_sequence ::=
  '[' [**parallel**[(chunk_specification)] [aspect_specification]]
    iterated_element_association ']'

4/5 reduction_attribute_designator ::=
  *reduction_*identifier(reduction_specification)

5/6 reduction_specification ::= *reducer_*name, *initial_value_*expression
 | [Reducer =>]*reducer_*name, Initial_Value => *initial_value_*expression

6/5 The iterated_element_association of a value_sequence shall not have a *key_*expression, nor shall it have a loop_parameter_specification that has the reserved word **reverse**.

7/5 The chunk_specification, if any, of a value_sequence shall be an *integer_*simple_expression.

Name Resolution Rules

8/5 The expected type for a reduction_attribute_reference shall be a single nonlimited type.

9/6 In the remainder of this subclause, we will refer to nonlimited subtypes *Value_Subtype* and *Accum_Subtype* of a reduction_attribute_reference. These subtypes and interpretations of the names and expressions of a reduction_attribute_reference are determined by the following rules:

10/6 - *Accum_Subtype* is a subtype of the expected type of the reduction_attribute_reference.

11/5 - A *reducer subprogram* is subtype conformant with one of the following specifications:

12/6
```
function Reducer(Accumulator : Accum_Subtype;
                 Value : Value_Subtype) return Accum_Subtype;
```

13/6
```
procedure Reducer(Accumulator : in out Accum_Subtype;
                  Value : in Value_Subtype);
```

14/5 - The *reducer_*name of a reduction_specification denotes a reducer subprogram.

- The expected type of an *initial_value*_expression of a reduction_specification is the type of *Accum_Subtype*. 15/6
- The expected type of the expression of the iterated_element_association of a value_sequence is the type of *Value_Subtype*. 16/6

Legality Rules

If the reduction_attribute_reference has a value_sequence with the reserved word **parallel**, *Accum_Subtype* and *Value_Subtype* shall statically match. 17/6

If the identifier of a reduction_attribute_designator is Parallel_Reduce, *Accum_Subtype* and *Value_Subtype* shall statically match. 18/6

Neither the type of *Accum_Subtype* nor *Value_Subtype* shall be an anonymous access type. 18.1/6

Static Semantics

A reduction_attribute_reference denotes a value, with its nominal subtype being *Accum_Subtype* (the subtype of the first parameter of the subprogram denoted by the *reducer*_name). 19/6

Dynamic Semantics

For the evaluation of a value_sequence, the iterated_element_association, the chunk_specification, and the aspect_specification, if any, are elaborated in an arbitrary order. Next an iteration is performed, and for each value conditionally produced by the iteration (see 5.5 and 5.5.2), the associated expression is evaluated with the loop parameter having this value, which produces a result that is converted to *Value_Subtype* and is used to define the next value in the sequence. 20/6

If the value_sequence does not have the reserved word **parallel**, it is produced as a single sequence of values by a single logical thread of control. If the reserved word **parallel** is present in the value_sequence, the enclosing reduction_attribute_reference is a parallel construct, and the sequence of values is generated by a parallel iteration (as defined in 5.5, 5.5.1, and 5.5.2), as a set of non-overlapping contiguous chunks (*subsequences*) with one logical thread of control (see Clause 9) associated with each subsequence. If there is a chunk_specification, it determines the maximum number of chunks, as defined in 5.5; otherwise the maximum number of chunks is implementation defined. 21/6

For a value_sequence V, the following attribute is defined: 22/5

V'Reduce(Reducer, Initial_Value) 23/5

This attribute represents a *reduction expression*, and is in the form of a reduction_-attribute_reference.

The evaluation of a use of this attribute begins by evaluating the parts of the reduction_-attribute_designator (the *reducer*_name Reducer and the *initial_value*_expression Initial_Value), in an arbitrary order. It then converts the the value of the *initial_value*_-expression (the *initial value*) to *Accum_Subtype* and then initializes the *accumulator* with the result. The value_sequence V is then evaluated. 24/6

If the value_sequence does not have the reserved word **parallel**, each value of the value_sequence is converted to *Value_Subtype* and then passed, in order, as the second (Value) parameter to a call on Reducer, with the first (Accumulator) parameter being the prior value of the accumulator, saving the result as the new value of the accumulator. The reduction expression yields the final value of the accumulator. 25/6

If the reserved word **parallel** is present in a value_sequence, then the (parallel) reduction expression is a parallel construct and the sequence has been partitioned into one or more subsequences (see above) each with its own separate logical thread of control. 26/5

27/6 Each logical thread of control creates a local accumulator for processing its subsequence. The accumulator for a subsequence is initialized to the result of converting the first value conditionally produced by the subsequence, if any, to *Accum_Subtype*, and calls on Reducer start with the second value of the subsequence (if any). The result for the subsequence, if non-empty, is the final value of its local accumulator.

28/6 After all logical threads of control of a parallel reduction expression have completed, Reducer is called for each non-empty subsequence, if any, in the original sequence order, passing the local accumulator for that subsequence as the second (Value) parameter, and the overall accumulator (initialized above to the initial value) as the first (Accumulator) parameter, with the result saved back in the overall accumulator. The parallel reduction expression yields the final value of the overall accumulator.

29/6 If the evaluation of the value_sequence yields an empty sequence of values, the reduction expression yields the initial value of the accumulator.

30/5 If an exception is propagated by one of the calls on Reducer, that exception is propagated from the reduction expression. If different exceptions are propagated in different logical threads of control, one is chosen arbitrarily to be propagated from the reduction expression as a whole.

31/5 For a prefix X of an array type (after any implicit dereference), or that denotes an iterable container object (see 5.5.1), the following attributes are defined:

32/5 X'Reduce(Reducer, Initial_Value)

X'Reduce is a reduction expression that yields a result equivalent to replacing the prefix of the attribute with the value_sequence:

32.1/5
```
[for Item of X => Item]
```

33/5 X'Parallel_Reduce(Reducer, Initial_Value)

X'Parallel_Reduce is a reduction expression that yields a result equivalent to replacing the attribute identifier with Reduce and the prefix of the attribute with the value_sequence:

33.1/5
```
[parallel for Item of X => Item]
```

Bounded (Run-Time) Errors

34/6 For a parallel reduction expression, it is a bounded error if the reducer subprogram is not associative. That is, for any arbitrary values of *Value_Subtype A*, *B*, *C* and a reducer function *R*, the result of *R* (*A*, *R* (*B*, *C*)) should produce a result equal to *R* (*R* (*A*, *B*), *C*)); it is a bounded error if *R* does not. The possible consequences are Program_Error, or a result that does not match the equivalent sequential reduction expression due to the order of calls on the reducer subprogram being unspecified in the overall reduction. Analogous rules apply in the case of a reduction procedure.

Examples

35/5 *Example of an expression function that returns its result as a reduction expression:*

36/5
```
function Factorial(N : Natural) return Natural is
   ([for J in 1..N => J]'Reduce("*", 1));
```

37/5 *Example of a reduction expression that computes the Sine of X using a Taylor expansion:*

38/5
```
function Sine (X : Float; Num_Terms : Positive := 5) return Float is
   ([for I in 1..Num_Terms =>
      (-1.0)**(I-1) * X**(2*I-1)/Float(Factorial(2*I-1))]
         'Reduce("+", 0.0));
```

39/5 *Example of a reduction expression that outputs the sum of squares:*

40/5
```
Put_Line ("Sum of Squares is" &
          Integer'Image([for I in 1 .. 10 => I**2]'Reduce("+", 0)));
```

Example of a reduction expression used to compute the value of Pi: 41/5

```
--  See 3.5.7.                                                          42/6
function Pi (Number_Of_Steps : Natural := 10_000) return Real is
  (1.0 / Real (Number_Of_Steps) *
    [for I in 1 .. Number_Of_Steps =>
        (4.0 / (1.0 + ((Real (I) - 0.5) *
           (1.0 / Real (Number_Of_Steps)))**2))]
              'Reduce(Reducer => "+", Initial_Value => 0.0));
```

Example of a reduction expression used to calculate the sum of elements of an array of integers: 43/5

```
A'Reduce("+", Initial_Value => 0)   -- See 4.3.3.                       44/6
```

Example of a reduction expression used to determine if all elements in a two-dimensional array of booleans are set to true: 45/5

```
Grid'Reduce("and", True)   -- See 3.6.                                  46/5
```

Example of a reduction expression used to calculate the minimum value of an array of integers in parallel: 47/5

```
A'Parallel_Reduce(Reducer => Integer'Min,                               48/6
                  Initial_Value => Integer'Last)
```

Example of a parallel reduction expression used to calculate the mean of the elements of a two-dimensional array of subtype Matrix (see 3.6) that are greater than 100.0: 49/5

```
type Accumulator is record                                              50/5
   Sum   : Real; -- See 3.5.7.
   Count : Integer;
end record;

function Accumulate (L, R : Accumulator) return Accumulator is          51/5
  (Sum   => L.Sum   + R.Sum,
   Count => L.Count + R.Count);

function Average_of_Values_Greater_Than_100 (M : Matrix) return Real is 52/5
   (declare
       Acc : constant Accumulator :=
          [parallel for Val of M when Val > 100.0 => (Val, 1)]
            'Reduce(Accumulate, (Sum => 0, Count => 0));
    begin
       Acc.Sum / Real(Acc.Count));
```

4.6 Type Conversions

Explicit type conversions, both value conversions and view conversions, are allowed between closely related types as defined below. This subclause also defines rules for value and view conversions to a particular subtype of a type, both explicit ones and those implicit in other constructs. 1/3

Syntax

type_conversion ::= 2
  subtype_mark(expression)
 | subtype_mark(name)

The *target subtype* of a type_conversion is the subtype denoted by the subtype_mark. The *operand* of a type_conversion is the expression or name within the parentheses; its type is the *operand type*. 3

One type is *convertible* to a second type if a type_conversion with the first type as operand type and the second type as target type is legal according to the rules of this subclause. Two types are convertible if each is convertible to the other. 4/3

5/5 A type_conversion is called a *view conversion* if both its target type and operand type are tagged, or if it appears in a call as an actual parameter of mode **out** or **in out**; other type_conversions are called *value conversions*.

Name Resolution Rules

6 The operand of a type_conversion is expected to be of any type.

7 The operand of a view conversion is interpreted only as a name; the operand of a value conversion is interpreted as an expression.

Legality Rules

8/2 In a view conversion for an untagged type, the target type shall be convertible (back) to the operand type.

Paragraphs 9 through 20 were reorganized and moved below.

21/3 If there is a type (other than a root numeric type) that is an ancestor of both the target type and the operand type, or both types are class-wide types, then at least one of the following rules shall apply:

21.1/2 • The target type shall be untagged; or

22 • The operand type shall be covered by or descended from the target type; or

23/2 • The operand type shall be a class-wide type that covers the target type; or

23.1/2 • The operand and target types shall both be class-wide types and the specific type associated with at least one of them shall be an interface type.

24/3 If there is no type (other than a root numeric type) that is the ancestor of both the target type and the operand type, and they are not both class-wide types, one of the following rules shall apply:

24.1/2 • If the target type is a numeric type, then the operand type shall be a numeric type.

24.2/2 • If the target type is an array type, then the operand type shall be an array type. Further:

24.3/2 • The types shall have the same dimensionality;

24.4/2 • Corresponding index types shall be convertible;

24.5/2 • The component subtypes shall statically match;

24.6/2 • If the component types are anonymous access types, then the accessibility level of the operand type shall not be statically deeper than that of the target type;

24.7/2 • Neither the target type nor the operand type shall be limited;

24.8/2 • If the target type of a view conversion has aliased components, then so shall the operand type; and

24.9/2 • The operand type of a view conversion shall not have a tagged, private, or volatile subcomponent.

24.10/2 • If the target type is *universal_access*, then the operand type shall be an access type.

24.11/2 • If the target type is a general access-to-object type, then the operand type shall be *universal_access* or an access-to-object type. Further, if the operand type is not *universal_access*:

24.12/2 • If the target type is an access-to-variable type, then the operand type shall be an access-to-variable type;

24.13/2 • If the target designated type is tagged, then the operand designated type shall be convertible to the target designated type;

- If the target designated type is class-wide, then the accessibility level of the operand designated (tagged) type shall not be statically deeper than the accessibility level of the target (access) type. 24.14/6
- If the target designated type is not tagged, then the designated types shall be the same, and either: 24.15/6
  - the designated subtypes shall statically match; or 24.16/6
  - the designated type shall be discriminated in its full view and unconstrained in any partial view, further: 24.17/6
    - at least one of the designated subtypes shall be unconstrained; 24.18/6
    - any predicate aspect specification that applies to one of the designated subtypes shall also apply to the other; and 24.19/6
    - the value of Object_Size (see 13.3) shall be the same for the designated subtypes if either has specified Object_Size to have a nonconfirming value; 24.20/6
- The accessibility level of the operand type shall not be statically deeper than that of the target type, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, the accessibility level of the operand type shall not be statically deeper than that of the declaration of the stand-alone object. 24.21/6

- If the target type is a pool-specific access-to-object type, then the operand type shall be *universal_access*. 24.22/6
- If the target type is an access-to-subprogram type, then the operand type shall be *universal_access* or an access-to-subprogram type. Further, if the operand type is not *universal_access*: 24.23/6
  - The designated profiles shall be subtype conformant. 24.24/6
  - The accessibility level of the operand type shall not be statically deeper than that of the target type. If the operand type is declared within a generic body, the target type shall be declared within the generic body. 24.25/6
  - If the target type has a Global aspect other than **in out all** or Unspecified, then each mode of the Global aspect of the operand type shall identify a subset of the variables identified by the corresponding mode of the target type Global aspect, or by the **in out** mode of the target type Global aspect. 24.26/6
  - If the target type is nonblocking, the operand type shall be nonblocking. 24.27/6

In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 24.28/6

Static Semantics

A type_conversion that is a value conversion denotes the value that is the result of converting the value of the operand to the target subtype. 25

A type_conversion that is a view conversion denotes a view of the object denoted by the operand. This view is a variable of the target type if the operand denotes a variable; otherwise, it is a constant of the target type. 26/3

The nominal subtype of a type_conversion is its target subtype. 27

Dynamic Semantics

For the evaluation of a type_conversion that is a value conversion, the operand is evaluated, and then the value of the operand is *converted* to a *corresponding* value of the target type, if any. If there is no value of 28

the target type that corresponds to the operand value, Constraint_Error is raised; this can only happen on conversion to a modular type, and only when the operand value is outside the base range of the modular type. Additional rules follow:

29 • Numeric Type Conversion

30 • If the target and the operand types are both integer types, then the result is the value of the target type that corresponds to the same mathematical integer as the operand.

31 • If the target type is a decimal fixed point type, then the result is truncated (toward 0) if the value of the operand is not a multiple of the *small* of the target type.

32 • If the target type is some other real type, then the result is within the accuracy of the target type (see G.2, “Numeric Performance Requirements”, for implementations that support the Numerics Annex).

33 • If the target type is an integer type and the operand type is real, the result is rounded to the nearest integer (away from zero if exactly halfway between two integers).

34 • Enumeration Type Conversion

35 • The result is the value of the target type with the same position number as that of the operand value.

36 • Array Type Conversion

37 • If the target subtype is a constrained array subtype, then a check is made that the length of each dimension of the value of the operand equals the length of the corresponding dimension of the target subtype. The bounds of the result are those of the target subtype.

38 • If the target subtype is an unconstrained array subtype, then the bounds of the result are obtained by converting each bound of the value of the operand to the corresponding index type of the target type. For each nonnull index range, a check is made that the bounds of the range belong to the corresponding index subtype.

39 • In either array case, the value of each component of the result is that of the matching component of the operand value (see 4.5.2).

39.1/2 • If the component types of the array types are anonymous access types, then a check is made that the accessibility level of the operand type is not deeper than that of the target type.

40 • Composite (Non-Array) Type Conversion

41 • The value of each nondiscriminant component of the result is that of the matching component of the operand value.

42 • The tag of the result is that of the operand. If the operand type is class-wide, a check is made that the tag of the operand identifies a (specific) type that is covered by or descended from the target type.

43 • For each discriminant of the target type that corresponds to a discriminant of the operand type, its value is that of the corresponding discriminant of the operand value; if it corresponds to more than one discriminant of the operand type, a check is made that all these discriminants are equal in the operand value.

44 • For each discriminant of the target type that corresponds to a discriminant that is specified by the derived_type_definition for some ancestor of the operand type (or if class-wide, some ancestor of the specific type identified by the tag of the operand), its value in the result is that specified by the derived_type_definition.

- For each discriminant of the operand type that corresponds to a discriminant that is specified by the derived_type_definition for some ancestor of the target type, a check is made that in the operand value it equals the value specified for it. 45
- For each discriminant of the result, a check is made that its value belongs to its subtype. 46

- Access Type Conversion 47
  - For an access-to-object type, a check is made that the accessibility level of the operand type is not deeper than that of the target type, unless the target type is an anonymous access type of a stand-alone object. If the target type is that of such a stand-alone object, a check is made that the accessibility level of the operand type is not deeper than that of the declaration of the stand-alone object; then if the check succeeds, the accessibility level of the target type becomes that of the operand type. 48/3
  - If the operand value is null, the result of the conversion is the null value of the target type. 49/2
  - If the operand value is not null, then the result designates the same object (or subprogram) as is designated by the operand value, but viewed as being of the target designated subtype (or profile); any checks associated with evaluating a conversion to the target designated subtype are performed. 50

After conversion of the value to the target type, if the target subtype is constrained, a check is performed that the value satisfies this constraint. If the target subtype excludes null, then a check is made that the value is not null. If predicate checks are enabled for the target subtype (see 3.2.4), a check is performed that the value satisfies the predicates of the target subtype, unless the conversion is:. 51/5

- a view conversion that is the target of an assignment statement and is not referenced with a target_name, or an actual parameter of mode **out**; or 51.1/5
- an implicit subtype conversion of an actual parameter of mode **out** to the nominal subtype of its formal parameter. 51.2/5

For the evaluation of a view conversion, the operand name is evaluated, and a new view of the object denoted by the operand is created, whose type is the target type; if the target type is composite, checks are performed as above for a value conversion. 52

The properties of this new view are as follows: 53

- If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each nondiscriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the target subtype is indefinite, or if the operand type is a descendant of the target type and has discriminants that were not inherited from the target type; 54/1
- If the target type is tagged, then an assignment to the view assigns to the corresponding part of the object denoted by the operand; otherwise, an assignment to the view assigns to the object, after converting the assigned value to the subtype of the object (which can raise Constraint_Error); 55/5
- Reading the value of the view yields the result of converting the value of the operand object to the target subtype (which can raise Constraint_Error), except if the object is of an elementary type and the view conversion is passed as an **out** parameter; in this latter case, the value of the operand object may be used to initialize the formal parameter without checking against any constraint of the target subtype (as described more precisely in 6.4.1). 56/5

If an Accessibility_Check fails, Program_Error is raised. If a predicate check fails, the effect is as defined in 3.2.4. Any other check associated with a conversion raises Constraint_Error if it fails. 57/5

58 Conversion to a type is the same as conversion to an unconstrained subtype of the type.

58.1/5 Evaluation of a value conversion of an object either creates a new anonymous object (similar to the object created by the evaluation of an aggregate or a function call) or yields a new view of the operand object without creating a new object:

58.2/4 • If the target type is a by-reference type and there is a type that is an ancestor of both the target type and the operand type then no new object is created;

58.3/4 • If the target type is an array type having aliased components and the operand type is an array type having unaliased components, then a new object is created;

58.4/5 • If the target type is an elementary type, then a new object is created;

58.5/5 • Otherwise, it is unspecified whether a new object is created.

58.6/5 If a new object is created, then the initialization of that object is an assignment operation.

59 NOTE 1 In addition to explicit type_conversions, type conversions are performed implicitly in situations where the expected type and the actual type of a construct differ, as is permitted by the type resolution rules (see 8.6). For example, an integer literal is of the type *universal_integer*, and is implicitly converted when assigned to a target of some specific integer type. Similarly, an actual parameter of a specific tagged type is implicitly converted when the corresponding formal parameter is of a class-wide type.

60 Even when the expected and actual types are the same, implicit subtype conversions are performed to adjust the array bounds (if any) of an operand to match the desired target subtype, or to raise Constraint_Error if the (possibly adjusted) value does not satisfy the constraints of the target subtype.

61/2 NOTE 2 A ramification of the overload resolution rules is that the operand of an (explicit) type_conversion cannot be an allocator, an aggregate, a string_literal, a character_literal, or an attribute_reference for an Access or Unchecked_Access attribute. Similarly, such an expression enclosed by parentheses is not allowed. A qualified_expression (see 4.7) can be used instead of such a type_conversion.

62 NOTE 3 The constraint of the target subtype has no effect for a type_conversion of an elementary type passed as an **out** parameter. Hence, it is recommended that the first subtype be specified as the target to minimize confusion (a similar recommendation applies to renaming and generic formal **in out** objects).

Examples

63 *Examples of numeric type conversion:*

64
```
Real(2*J)       -- value is converted to floating point
Integer(1.6)    -- value is 2
Integer(-0.4)   -- value is 0
```

65 *Example of conversion between derived types:*

66
```
type A_Form is new B_Form;
```

67
```
X : A_Form;
Y : B_Form;
```

68
```
X := A_Form(Y);
Y := B_Form(X);   -- the reverse conversion
```

69 *Examples of conversions between array types:*

70
```
type Sequence is array (Integer range <>) of Integer;
subtype Dozen is Sequence(1 .. 12);
Ledger : array(1 .. 100) of Integer;
```

71
```
Sequence(Ledger)             -- bounds are those of Ledger
Sequence(Ledger(31 .. 42))   -- bounds are 31 and 42
Dozen(Ledger(31 .. 42))      -- bounds are those of Dozen
```

4.7 Qualified Expressions

A qualified_expression is used to state explicitly the type, and to verify the subtype, of an operand that is either an expression or an aggregate. 1

Syntax

qualified_expression ::= 2
  subtype_mark'(expression) | subtype_mark'aggregate

Name Resolution Rules

The expected type for the *operand* (the expression or aggregate) is determined by the subtype_mark. Furthermore, the operand shall resolve to be either the specified expected type or a universal type that covers it. 3/5

Static Semantics

If the operand of a qualified_expression denotes an object, the qualified_expression denotes a constant view of that object. The nominal subtype of a qualified_expression is the subtype denoted by the subtype_mark. 3.1/3

Dynamic Semantics

The evaluation of a qualified_expression evaluates the operand (and if of a universal type, converts it to the type determined by the subtype_mark) and checks that its value belongs to the subtype denoted by the subtype_mark. The exception Constraint_Error is raised if this check fails. Furthermore, if predicate checks are enabled for the subtype denoted by the subtype_mark, a check is performed as defined in 3.2.4 that the value satifies the predicates of the subtype. 4/5

NOTE When a given context does not uniquely identify an expected type, a qualified_expression can be used to do so. In particular, if an overloaded name or aggregate is passed to an overloaded subprogram, it can be necessary to qualify the operand to resolve its type. 5/5

Examples

Examples of disambiguating expressions using qualification: 6

```
type Mask is (Fix, Dec, Exp, Signif);                                                    7
type Code is (Fix, Cla, Dec, Tnz, Sub);

Print (Mask'(Dec));  -- Dec is of type Mask                                               8
Print (Code'(Dec));  -- Dec is of type Code

for J in Code'(Fix) .. Code'(Dec) loop ... -- qualification is necessary for either Fix or Dec   9/5
for J in Code range Fix .. Dec loop ...    -- qualification unnecessary
for J in Code'(Fix) .. Dec loop ...        -- qualification unnecessary for Dec

Dozen'(1 | 3 | 5 | 7 => 2, others => 0) -- see 4.6                                      10
```

4.8 Allocators

1 The evaluation of an allocator creates an object and yields an access value that designates the object.

Syntax

2/3 allocator ::=
new [subpool_specification] subtype_indication
| **new** [subpool_specification] qualified_expression

2.1/3 subpool_specification ::= (*subpool_handle*_name)

2.2/3 For an allocator with a subtype_indication, the subtype_indication shall not specify a null_exclusion.

Name Resolution Rules

3/3 The expected type for an allocator shall be a single access-to-object type with designated type *D* such that either *D* covers the type determined by the subtype_mark of the subtype_indication or qualified_expression, or the expected type is anonymous and the determined type is *D*'Class. A *subpool_handle*_name is expected to be of any type descended from Subpool_Handle, which is the type used to identify a subpool, declared in package System.Storage_Pools.Subpools (see 13.11.4).

Legality Rules

4 An *initialized* allocator is an allocator with a qualified_expression. An *uninitialized* allocator is one with a subtype_indication. In the subtype_indication of an uninitialized allocator, a constraint is permitted only if the subtype_mark denotes an unconstrained composite subtype; if there is no constraint, then the subtype_mark shall denote a definite subtype.

5/2 If the type of the allocator is an access-to-constant type, the allocator shall be an initialized allocator.

5.1/3 If a subpool_specification is given, the type of the storage pool of the access type shall be a descendant of Root_Storage_Pool_With_Subpools.

5.2/3 If the designated type of the type of the allocator is class-wide, the accessibility level of the type determined by the subtype_indication or qualified_expression shall not be statically deeper than that of the type of the allocator.

5.3/3 If the subtype determined by the subtype_indication or qualified_expression of the allocator has one or more access discriminants, then the accessibility level of the anonymous access type of each access discriminant shall not be statically deeper than that of the type of the allocator (see 3.10.2).

5.4/3 An allocator shall not be of an access type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero.

5.5/3 If the designated type of the type of the allocator is limited, then the allocator shall not be used to define the value of an access discriminant, unless the discriminated type is immutably limited (see 7.5).

5.6/3 In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit.

Static Semantics

6/3 If the designated type of the type of the allocator is elementary, then the subtype of the created object is the designated subtype. If the designated type is composite, then the subtype of the created object is the

designated subtype when the designated subtype is constrained or there is an ancestor of the designated type that has a constrained partial view; otherwise, the created object is constrained by its initial value (even if the designated subtype is unconstrained with defaults).

Dynamic Semantics

For the evaluation of an initialized allocator, the evaluation of the qualified_expression is performed first. An object of the designated type is created and the value of the qualified_expression is converted to the designated subtype and assigned to the object. 7/2

For the evaluation of an uninitialized allocator, the elaboration of the subtype_indication is performed first. Then: 8

- If the designated type is elementary, an object of the designated subtype is created and any implicit initial value is assigned; 9/2
- If the designated type is composite, an object of the designated type is created with tag, if any, determined by the subtype_mark of the subtype_indication. This object is then initialized by default (see 3.3.1) using the subtype_indication to determine its nominal subtype. A check is made that the value of the object belongs to the designated subtype. Constraint_Error is raised if this check fails. This check and the initialization of the object are performed in an arbitrary order. 10/2

For any allocator, if the designated type of the type of the allocator is class-wide, then a check is made that the master of the type determined by the subtype_indication, or by the tag of the value of the qualified_expression, includes the elaboration of the type of the allocator. Similarly, for an allocator that defines the result of a function with an anonymous access-to-tagged-type result, a check is made that the master of the type determined by the subtype_indication, or by the tag of the value of the qualified_expression, includes the master of the object created by the allocator (see 3.10.2). If any part of the subtype determined by the subtype_indication or qualified_expression of the allocator (or by the tag of the value if the type of the qualified_expression is class-wide) has one or more access discriminants, then a check is made that the accessibility level of the anonymous access type of each access discriminant is not deeper than that of the type of the allocator. Program_Error is raised if any of these checks fail. 10.1/6

If the object to be created by an allocator has a controlled or protected part, and the finalization of the collection of the type of the allocator (see 7.6.1) has started, Program_Error is raised. 10.2/2

If the object to be created by an allocator contains any tasks, and the master of the type of the allocator is completed, and all of the dependent tasks of the master are terminated (see 9.3), then Program_Error is raised. 10.3/2

If the allocator includes a *subpool_handle*_name, Constraint_Error is raised if the subpool handle is **null**. Program_Error is raised if the subpool does not *belong* (see 13.11.4) to the storage pool of the access type of the allocator. 10.4/3

If the created object contains any tasks, they are activated (see 9.2). Finally, an access value that designates the created object is returned. 11

Bounded (Run-Time) Errors

It is a bounded error if the finalization of the collection of the type (see 7.6.1) of the allocator has started. If the error is detected, Program_Error is raised. Otherwise, the allocation proceeds normally. 11.1/2

NOTE 1 Allocators cannot create objects of an abstract type. See 3.9.3. 12

NOTE 2 If any part of the created object is controlled, the initialization includes calls on corresponding Initialize or Adjust procedures. See 7.6. 13

14/5 NOTE 3 As explained in 13.11, “Storage Management”, the storage for an object allocated by an allocator comes from a storage pool (possibly user defined). The exception Storage_Error is raised by an allocator if there is not enough storage. Instances of Unchecked_Deallocation can be used to explicitly reclaim storage.

15/5 NOTE 4 Implementations can, if desired, provide garbage collection.

Examples

16 *Examples of allocators:*

17
```
new Cell'(0, null, null)                    -- initialized explicitly, see 3.10.1
new Cell'(Value => 0, Succ => null, Pred => null)
                                            -- initialized explicitly
new Cell                                    -- not initialized
```

18
```
new Matrix(1 .. 10, 1 .. 20)                -- the bounds only are given
new Matrix'(1 .. 10 => (1 .. 20 => 0.0))    -- initialized explicitly
```

19
```
new Buffer(100)                             -- the discriminant only is given
new Buffer'(Size => 80, Pos => 0, Value => (1 .. 80 => 'A'))
                                            -- initialized explicitly
```

20
```
Expr_Ptr'(new Literal)
                    -- allocator for access-to-class-wide type, see 3.9.1
Expr_Ptr'(new Literal'(Expression with 3.5)) -- initialized explicitly
```

4.9 Static Expressions and Static Subtypes

1 Certain expressions of a scalar or string type are defined to be static. Similarly, certain discrete ranges are defined to be static, and certain scalar and string subtypes are defined to be static subtypes. *Static* means determinable at compile time, using the declared properties or values of the program entities.

Static Semantics

2 A static expression is a scalar or string expression that is one of the following:

3/5 • a numeric_literal of a numeric type;

4 • a string_literal of a static string subtype;

5/5 • a name that denotes the declaration of a static constant;

5.1/5 • a name that denotes a named number, and that is interpreted as a value of a numeric type;

6 • a function_call whose *function_*name or *function_*prefix statically denotes a static function, and whose actual parameters, if any (whether given explicitly or by default), are all static expressions;

7 • an attribute_reference that denotes a scalar value, and whose prefix denotes a static scalar subtype;

8/5 • an attribute_reference whose prefix statically names a statically constrained array object or array subtype, and whose attribute_designator is First, Last, or Length, with an optional dimension;

8.1/6 •

9/5 • a type_conversion whose subtype_mark denotes a static (scalar or string) subtype, and whose operand is a static expression;

10 • a qualified_expression whose subtype_mark denotes a static (scalar or string) subtype, and whose operand is a static expression;

11/4 • a membership test whose *tested_*simple_expression is a static expression, and whose membership_choice_list consists only of membership_choices that are either static

*choice*_simple_expressions, static ranges, or subtype_marks that denote a static (scalar or string) subtype;

- a short-circuit control form both of whose relations are static expressions; 12
- a conditional_expression all of whose conditions, *selecting*_expressions, and *dependent*_expressions are static expressions; 12.1/3
- a declare_expression whose *body*_expression is static and each of whose declarations, if any, is either the declaration of a static constant or is an object_renaming_declaration with an *object*_name that statically names the renamed object; 12.2/5
- a static expression enclosed in parentheses. 13

A name *statically denotes* an entity if it denotes the entity and: 14

- It is a direct_name, expanded name, or character_literal, and it denotes a declaration other than a renaming_declaration; or 15
- It is an attribute_reference whose prefix statically denotes some entity; or 16
- It is a target_name (see 5.2.1) in an assignment_statement whose *variable*_name statically denotes some entity; or 16.1/5
- It denotes a renaming_declaration with a name that statically denotes the renamed entity. 17

A name *statically names* an object if it: 17.1/5

- statically denotes the declaration of an object (possibly through one or more renames); 17.2/5
- is a target_name in an assignment_statement whose *variable*_name statically names some entity; 17.3/6
- denotes a renaming_declaration with a name that statically names the renamed entity; 17.4/6
- is a selected_component whose prefix statically names an object, there is no implicit dereference of the prefix, and the selector_name does not denote a component_declaration occurring within a variant_part; or 17.5/6
- is an indexed_component whose prefix statically names an object, there is no implicit dereference of the prefix, the object is statically constrained, and the index expressions of the object are static and have values that are within the range of the index constraint. 17.6/6

For an entity other than an object, a name statically names an entity if the name statically denotes the entity. 17.7/6

A *static function* is one of the following: 18

- a predefined operator whose parameter and result types are all scalar types none of which are descendants of formal scalar types; 19
- a predefined relational operator whose parameters are of a string type that is not a descendant of a formal array type; 19.1/5
- a predefined concatenation operator whose result type is a string type that is not a descendant of a formal array type; 20/5
- a shifting or rotating function associated with a modular type declared in package Interfaces (see B.2); 20.1/5
- an enumeration literal; 21
- a static expression function (see 6.8); 21.1/5

22 • a language-defined attribute that is a function, if the prefix denotes a static scalar subtype, and if the parameter and result types are scalar.

23 In any case, a generic formal subprogram is not a static function.

24/5 A *static constant* is a constant view declared by a full constant declaration or an object_renaming_declaration with a static nominal subtype, having a value defined by a static scalar expression or by a static string expression, and which satisfies any constraint or predicate that applies to the nominal subtype.

25 A *static range* is a range whose bounds are static expressions, or a range_attribute_reference that is equivalent to such a range. A *static discrete_range* is one that is a static range or is a subtype_indication that defines a static scalar subtype. The base range of a scalar type is a static range, unless the type is a descendant of a formal scalar type.

26/3 A *static subtype* is either a *static scalar subtype* or a *static string subtype*. A static scalar subtype is an unconstrained scalar subtype whose type is not a descendant of a formal type, or a constrained scalar subtype formed by imposing a compatible static constraint on a static scalar subtype. A static string subtype is an unconstrained string subtype whose index subtype and component subtype are static, or a constrained string subtype formed by imposing a compatible static constraint on a static string subtype. In any case, the subtype of a generic formal object of mode **in out**, and the result subtype of a generic formal function, are not static. Also, a subtype is not static if any Dynamic_Predicate specifications apply to it.

27 The different kinds of *static constraint* are defined as follows:

28 • A null constraint is always static;

29 • A scalar constraint is static if it has no range_constraint, or one with a static range;

30 • An index constraint is static if each discrete_range is static, and each index subtype of the corresponding array type is static;

31 • A discriminant constraint is static if each expression of the constraint is static, and the subtype of each discriminant is static.

31.1/2 In any case, the constraint of the first subtype of a scalar formal type is neither static nor null.

32 A subtype is *statically constrained* if it is constrained, and its constraint is static. An object is *statically constrained* if its nominal subtype is statically constrained, or if it is a static string constant.

Legality Rules

32.1/3 An expression is *statically unevaluated* if it is part of:

32.2/3 • the right operand of a static short-circuit control form whose value is determined by its left operand; or

32.3/3 • a *dependent_*expression of an if_expression whose associated condition is static and equals False; or

32.4/3 • a condition or *dependent_*expression of an if_expression where the condition corresponding to at least one preceding *dependent_*expression of the if_expression is static and equals True; or

32.5/3 • a *dependent_*expression of a case_expression whose *selecting_*expression is static and whose value is not covered by the corresponding discrete_choice_list; or

32.6/4 • a *choice_*simple_expression (or a simple_expression of a range that occurs as a membership_choice of a membership_choice_list) of a static membership test that is preceded in the enclosing membership_choice_list by another item whose individual membership test (see 4.5.2) statically yields True.

A static expression is evaluated at compile time except when it is statically unevaluated. The compile-time evaluation of a static expression is performed exactly, without performing Overflow_Checks. For a static expression that is evaluated: 33/3

- The expression is illegal if its evaluation fails a language-defined check other than Overflow_Check. For the purposes of this evaluation, the assertion policy is assumed to be Check. 34/3
- If the expression is not part of a larger static expression and the expression is expected to be of a single specific type, then its value shall be within the base range of its expected type. Otherwise, the value may be arbitrarily large or small. 35/2
- If the expression is of type *universal_real* and its expected type is a decimal fixed point type, then its value shall be a multiple of the *small* of the decimal type. This restriction does not apply if the expected type is a descendant of a formal scalar type (or a corresponding actual type in an instance). 36/2

In addition to the places where Legality Rules normally apply (see 12.3), the above restrictions also apply in the private part of an instance of a generic unit. 37/2

Implementation Requirements

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the implementation shall round or truncate the value (according to the Machine_Rounds attribute of the expected type) to the nearest machine number of the expected type; if the value is exactly half-way between two machine numbers, the rounding performed is implementation-defined. If the expected type is a descendant of a formal type, or if the static expression appears in the body of an instance of a generic unit and the corresponding expression is nonstatic in the corresponding generic body, then no special rounding or truncating is required — normal accuracy rules apply (see Annex G). 38/2

Implementation Advice

For a real static expression that is not part of a larger static expression, and whose expected type is not a descendant of a formal type, the rounding should be the same as the default rounding for the target system. 38.1/2

NOTE 1 An expression can be static even if it occurs in a context where staticness is not required. 39

NOTE 2 A static (or run-time) type_conversion from a real type to an integer type performs rounding. If the operand value is exactly half-way between two integers, the rounding is performed away from zero. 40

Examples

Examples of static expressions: 41

```
1 + 1        -- 2                                                    42
abs(-10)*3   -- 30

Kilo : constant := 1000;                                             43
Mega : constant := Kilo*Kilo;    -- 1_000_000
Long : constant := Float'Digits*2;

Half_Pi     : constant := Pi/2;             -- see 3.3.2             44
Deg_To_Rad : constant := Half_Pi/90;
Rad_To_Deg : constant := 1.0/Deg_To_Rad;
                         -- equivalent to 1.0/((3.14159_26536/2)/90)
```

4.9.1 Statically Matching Constraints and Subtypes

Static Semantics

A constraint *statically matches* another constraint if: 1/2

1.1/2 • both are null constraints;

1.2/2 • both are static and have equal corresponding bounds or discriminant values;

1.3/2 • both are nonstatic and result from the same elaboration of a constraint of a subtype_indication or the same evaluation of a range of a discrete_subtype_definition; or

1.4/2 • both are nonstatic and come from the same formal_type_declaration.

1.5/5 The Global or Global'Class aspects (see 6.1.2) of two entities *statically match* if both consist of a single global_aspect_definition where each is the reserved word **null**, or each is of the form "global_mode global_designator" with each global_mode being the same sequence of reserved words and each global_designator being the same reserved word, or each being a global_name that statically names the same entity.

2/5 A subtype *statically matches* another subtype of the same type if they have statically matching constraints, all predicate specifications that apply to them come from the same declarations, Nonblocking aspects have the same value, global aspects statically match, Object_Size (see 13.3) has been specified to have a nonconfirming value for either both or neither, and the nonconfirming values, if any, are the same, and, for access subtypes, either both or neither exclude null. Two anonymous access-to-object subtypes statically match if their designated subtypes statically match, and either both or neither exclude null, and either both or neither are access-to-constant. Two anonymous access-to-subprogram subtypes statically match if their designated profiles are subtype conformant, and either both or neither exclude null.

3 Two ranges of the same type *statically match* if both result from the same evaluation of a range, or if both are static and have equal corresponding bounds.

4/3 A constraint is *statically compatible* with a scalar subtype if it statically matches the constraint of the subtype, or if both are static and the constraint is compatible with the subtype. A constraint is *statically compatible* with an access or composite subtype if it statically matches the constraint of the subtype, or if the subtype is unconstrained.

5/3 Two statically matching subtypes are statically compatible with each other. In addition, a subtype *S1* is statically compatible with a subtype *S2* if:

6/3 • the constraint of *S1* is statically compatible with *S2*, and

7/3 • if *S2* excludes null, so does *S1*, and

8/3 • either:

9/3 • all predicate specifications that apply to *S2* apply also to *S1*, or

10/4 • both subtypes are static, every value that satisfies the predicates of *S1* also satisfies the predicates of *S2*, and it is not the case that both types each have at least one applicable predicate specification, predicate checks are enabled (see 11.4.2) for *S2*, and predicate checks are not enabled for *S1*.

4.10 Image Attributes

1/5 An *image* of a value is a string representing the value in display form. The attributes Image, Wide_Image, and Wide_Wide_Image are available to produce the image of a value as a String, Wide_String, or Wide_Wide_String (respectively). User-defined images for a given type can be implemented by overriding the default implementation of the attribute Put_Image.

Static Semantics

For every subtype S of a type T other than *universal_real* or *universal_fixed*, the following type-related operational attribute is defined: 2/5

S'Put_Image S'Put_Image denotes a procedure with the following specification: 3/5

```
procedure S'Put_Image                                                    3.1/5
   (Buffer : in out
                Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
    Arg    : in T);
```

The default implementation of S'Put_Image writes (using Wide_Wide_Put) an *image* of the value of *Arg*. 3.2/5

The Put_Image attribute may be specified for any specific noninterface type T either via an attribute_definition_clause or via an aspect_specification specifying the Put_Image aspect of the type. The Put_Image aspect is not inherited, but rather is implicitly composed for derived types, as defined below. 4/6

For an aspect_specification or attribute_definition_clause specifying Put_Image, the subtype of the *Arg* parameter shall be the first subtype or the base subtype if scalar, and the first subtype if not scalar. 5/5

The behavior of the default implementation of S'Put_Image depends on the class of T. 6/5

For an untagged derived type declared without a known_discriminant_part, or a null extension, the default implementation of T'Put_Image invokes the Put_Image for its parent type on a conversion of the parameter of type T to the parent type. 7/6

For a nonderived elementary type, the implementation is equivalent to: 8/5

```
procedure Scalar_Type'Put_Image                                          9/5
  (Buffer : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Arg    : in Scalar_Type) is
begin
   Buffer.Wide_Wide_Put (<described below>);
end Scalar_Type'Put_Image;
```

where the Wide_Wide_String value written out to the text buffer is defined as follows: 10/5

- For an integer type, the image written out is the corresponding decimal literal, without underlines, leading zeros, exponent, or trailing spaces, but with a single leading character that is either a minus sign or a space. 11/5
- For an enumeration type, the image written out is either the corresponding identifier in upper case or the corresponding character literal (including the two apostrophes); neither leading nor trailing spaces are included. For a *nongraphic character* (a value of a character type that has no enumeration literal associated with it), the value is a corresponding language-defined name in upper case (for example, the image of the nongraphic character identified as *nul* is `"NUL"` — the quotes are not part of the image). 12/5
- For a floating point type, the image written out is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either a minus sign or a space, a single digit (that is nonzero unless the value is zero), a decimal point, S'Digits-1 (see 3.5.8) digits after the decimal point (but one if S'Digits is one), an upper case E, the sign of the exponent (either + or -), and two or more digits (with leading zeros if necessary) representing the exponent. If S'Signed_Zeros is True, then the leading character is a minus sign for a negatively signed zero. 13/5
- For a fixed point type, the image written out is a decimal real literal best approximating the value (rounded away from zero if halfway between) with a single leading character that is either 14/5

a minus sign or a space, one or more digits before the decimal point (with no redundant leading zeros), a decimal point, and S'Aft (see 3.5.10) digits after the decimal point.

15/5 • For an access type (named or anonymous), the image written out depends on whether the value is **null**. If it is **null**, then the image is `"NULL"`. Otherwise the image is a left parenthesis followed by `"ACCESS"`, a space, and a sequence of graphic characters, other than space or right parenthesis, representing the location of the designated object, followed by a right parenthesis, as in `"(ACCESS FF0012AC)"`.

16/6 For a nonnull type extension, or an untagged derived type declared with a known_discriminant_part, the default implementation of T'Put_Image depends on whether there exists a noninterface ancestor of T (other than T itself) for which the Put_Image aspect has been directly specified (visibly or privately). If so, then:

16.1/6 • if T is a nonnull type extension, T'Put_Image will generate an image based on extension aggregate syntax where the ancestor type of the extension aggregate is the nearest ancestor type whose Put_Image aspect has been specified;

16.2/6 • if T is an untagged derived type with a known_discriminant_part, T'Put_Image invokes the Put_Image for its parent type on a conversion of the parameter of type T to the parent type.

16.3/6 If no such ancestor exists, then the default implementation of T'Put_Image is the same as described below for a nonderived record or protected type.

17/5 For a specific, nonderived composite type:

18/6 • *This paragraph was deleted.*

19/6 • For an array type T, the default implementation of T'Put_Image generates an image based on named (not positional) array aggregate syntax (with '[' and ']' as the delimiters) using calls to the Put_Image procedures of the index type(s) and the element type to generate images for values of those types. The components are written in increasing index order at each level, using a nested syntax for a multidimensional array, as in a multidimensional array aggregate.

20/5 The case of a null array is handled specially, using ranges for index bounds and `"<>"` as a syntactic component-value placeholder.

21/6 • For a record type (or, as indicated above, certain composite derived types), or a protected type, the default implementation of T'Put_Image generates an image based on named (not positional) record aggregate syntax (except that for a protected type, the initial left parenthesis is followed by `"PROTECTED with "`). Component names are displayed in upper case, following the rules for the image of an enumeration value. Component values are displayed via calls to the Put_Image procedure of the component type. Components are displayed in the order they occur within the declarative region of the type (see 8.1 and 3.4).

22/5 The image written out for a record having no components (including any interface type) is `"(NULL RECORD)"`. The image written out for a componentless protected type is `"(PROTECTED NULL RECORD)"`. In the case of a protected type T, a call to the default implementation of T'Put_Image begins only one protected (read-only) action.

23/5 • For an undiscriminated task type, the default implementation of T'Put_Image generates an image of the form `"(TASK <task_id_image>)"` where <task_id_image> is the result obtained by calling Task_Identification.Image with the id of the given task and then passing that String to Characters.Conversions.To_Wide_Wide_String.

24/6 • For a discriminated task type, the default implementation of T'Put_Image also includes discriminant values, in the order they occur within the known_discriminant_part, as in:

25/5 `"(TASK <task_id_image> with D1 => 123, D2 => 456)"`

For a class-wide type, the default implementation of T'Put_Image generates an image based on qualified expression syntax. Wide_Wide_Put is called with Wide_Wide_Expanded_Name of *Arg*'Tag. Then S'Put_Image is called, where S is the specific type identified by *Arg*'Tag. 26/5

T'Put_Image is the same for both the partial view and full view of T, if T has a partial view. 27/5

In the parameter_and_result_profile for the default implementation of Put_Image, the subtype of the *Arg* parameter is the base subtype of *T* if *T* is a scalar type, and the first subtype otherwise. For an aspect_specification or attribute_definition_clause specifying Put_Image, the subprogram name shall denote a nonabstract procedure whose second parameter is either of the first subtype of *T*, or as an option when *T* is scalar, the base subtype of *T*. 28/5

For every subtype S of a type T, the following attributes are defined: 29/5

S'Wide_Wide_Image 30/5

S'Wide_Wide_Image denotes a function with the following specification:

```
function S'Wide_Wide_Image(Arg : S'Base)
  return Wide_Wide_String
```
30.1/5

S'Wide_Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Wide_Wide_Get. The lower bound of the result is one. Any exception propagated by the call of S'Put_Image is propagated. 30.2/5

S'Wide_Image 31/5

S'Wide_Image denotes a function with the following specification:

```
function S'Wide_Image(Arg : S'Base)
  return Wide_String
```
32/5

S'Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Wide_Get. The lower bound of the result is one. Any exception propagated by the call of S'Put_Image is propagated. 33/5

S'Image S'Image denotes a function with the following specification: 34/5

```
function S'Image(Arg : S'Base)
  return String
```
35/5

S'Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Get. The lower bound of the result is one. Any exception propagated by the call of S'Put_Image is propagated. 36/5

For a prefix X of a type T other than *universal_real* or *universal_fixed*, the following attributes are defined: 37/5

X'Wide_Wide_Image 38/5

X'Wide_Wide_Image denotes the result of calling function S'Wide_Wide_Image with *Arg* being X, where S is the nominal subtype of X.

X'Wide_Image 39/5

X'Wide_Image denotes the result of calling function S'Wide_Image with *Arg* being X, where S is the nominal subtype of X.

X'Image X'Image denotes the result of calling function S'Image with *Arg* being X, where S is the nominal subtype of X. 40/5

Implementation Permissions

41/5 An implementation may transform the image generated by the default implementation of S'Put_Image for a composite subtype S in the following ways:

42/5 • If S is a composite subtype, the leading character of the image *M* of a component value or index value is a space, and the immediately preceding character (if any) is an open parenthesis, open bracket, or space, then the leading space of the image *M* may be omitted.

43/5 • If S is an array subtype, the low bound of the array in each dimension equals the low bound of the corresponding index subtype, and the array value is not a null array value, then positional array aggregate syntax may be used.

44/5 • If S is an array subtype and the given value can be displayed using named_array_aggregate syntax where some discrete_choice_list identifies more than one index value by identifying a sequence of one or more ranges and values separated by vertical bars, then this image may be generated instead; this may involve the reordering of component values.

45/5 • Similarly, if S is a record subtype (or a discriminated type) and the given value can be displayed using named component association syntax where the length of some component_choice_list is greater than one, then this image may be generated instead; this may involve the reordering of component values.

46/5 • Additional spaces (Wide_Wide_Characters with position 32), and calls to the New_Line operation of a text buffer, may be inserted to improve readability of the generated image, with the spaces inserted directly or via use of the Increase_Indent and Decrease_Indent procedures.

47/5 • For a string type, implementations may produce an image corresponding to a string literal.

48/5 • For an unchecked union type, implementations may raise Program_Error or produce some recognizable image (such as "`(UNCHECKED UNION)`") that does not require reading the discriminants.

49/5 For each language-defined nonscalar type T, T'Put_Image may be specified.

Implementation Requirements

50/5 For each language-defined container type T (that is, each of the Vector, List, Map, Set, Tree, and Holder types defined in the various children of Ada.Containers), T'Put_Image shall be specified so that T'Image produces a result consistent with array aggregate syntax (using '[' and ']' as delimiters) as follows:

51/5 • Vector images shall be consistent with the default image of an array type with the same index and component types.

52/5 • Map images shall be consistent with named array aggregate syntax, using key value images in place of discrete choice names. For example, `[Key1 => Value1, Key2 => Value2]`.

53/5 • Set, List, and Holder images shall be consistent with positional array aggregate syntax. List elements shall occur in order within an image of a list. The image of an empty holder shall be `[]`.

54/5 • Tree images (and images of subtrees of trees) shall be consistent with positional array aggregate syntax. For example, `[[1, 2], [111, 222, 333]]`.

55/5 For each language-defined nonscalar type T that has a primitive language-defined Image function whose profile is type conformant with that of T'Image (for example, Ada.Numerics.Float_Random.State has such an Image function), T'Put_Image shall be specified so that T'Image yields the same result as that Image function.

Implementation Advice

For each language-defined private type T, T'Image should generate an image that would be meaningful based only on the relevant public interfaces, as opposed to requiring knowledge of the implementation of the private type. 56/5

5 Statements

A statement defines an action to be performed upon its execution. 1

This clause describes the general rules applicable to all statements. Some statements are discussed in later clauses: Procedure_call_statements and return statements are described in Clause 6, “Subprograms”. Entry_call_statements, requeue_statements, delay_statements, accept_statements, select_statements, and abort_statements are described in Clause 9, “Tasks and Synchronization”. Raise_statements are described in Clause 11, “Exceptions”, and code_statements in Clause 13, “Representation Issues”. The remaining forms of statements are presented in this clause. 2/5

5.1 Simple and Compound Statements - Sequences of Statements

A statement is either simple or compound. A simple_statement encloses no other statement. A compound_statement can enclose simple_statements and other compound_statements. A *parallel construct* is a construct that introduces additional logical threads of control (see Clause 9) without creating a new task. Parallel loops (see 5.5) and parallel_block_statements (see 5.6.1) are parallel constructs. 1/5

Syntax

sequence_of_statements ::= statement {statement} {label} 2/3

statement ::= 3
  {label} simple_statement | {label} compound_statement

simple_statement ::= null_statement 4/2
  | assignment_statement | exit_statement
  | goto_statement | procedure_call_statement
  | simple_return_statement | entry_call_statement
  | requeue_statement | delay_statement
  | abort_statement | raise_statement
  | code_statement

compound_statement ::= 5/5
    if_statement | case_statement
  | loop_statement | block_statement
  | extended_return_statement
  | parallel_block_statement
  | accept_statement | select_statement

null_statement ::= **null**; 6

label ::= <<*label*_statement_identifier>> 7

statement_identifier ::= direct_name 8

The direct_name of a statement_identifier shall be an identifier (not an operator_symbol). 9

Name Resolution Rules

The direct_name of a statement_identifier shall resolve to denote its corresponding implicit declaration (see below). 10

Legality Rules

11 Distinct identifiers shall be used for all statement_identifiers that appear in the same body, including inner block_statements but excluding inner program units.

Static Semantics

12 For each statement_identifier, there is an implicit declaration (with the specified identifier) at the end of the declarative_part of the innermost block_statement or body that encloses the statement_identifier. The implicit declarations occur in the same order as the statement_identifiers occur in the source text. If a usage name denotes such an implicit declaration, the entity it denotes is the label, loop_statement, or block_statement with the given statement_identifier.

12.1/3 If one or more labels end a sequence_of_statements, an implicit null_statement follows the labels before any following constructs.

Dynamic Semantics

13 The execution of a null_statement has no effect.

14/2 A *transfer of control* is the run-time action of an exit_statement, return statement, goto_statement, or requeue_statement, selection of a terminate_alternative, raising of an exception, or an abort, which causes the next action performed to be one other than what would normally be expected from the other rules of the language. As explained in 7.6.1, a transfer of control can cause the execution of constructs to be completed and then left, which may trigger finalization.

15 The execution of a sequence_of_statements consists of the execution of the individual statements in succession until the sequence_ is completed.

16/5 Within a parallel construct, if a transfer of control out of the construct is initiated by one of the logical threads of control, an attempt is made to *cancel* all other logical threads of control initiated by the parallel construct. Once all other logical threads of control of the construct either complete or are canceled, the transfer of control occurs. If two or more logical threads of control of the same construct initiate such a transfer of control concurrently, one of them is chosen arbitrarily and the others are canceled.

17/5 When a logical thread of control is canceled, the cancellation causes it to complete as though it had performed a transfer of control to the point where it would have finished its execution. Such a cancellation is deferred while the logical thread of control is executing within an abort-deferred operation (see 9.8), and may be deferred further, but not past a point where the logical thread initiates a new nested parallel construct or reaches an exception handler that is outside such an abort-deferred operation.

Bounded (Run-Time) Errors

18/5 During the execution of a parallel construct, it is a bounded error to invoke an operation that is potentially blocking (see 9.5). Program_Error is raised if the error is detected by the implementation; otherwise, the execution of the potentially blocking operation can either proceed normally, or it can result in the indefinite blocking of some or all of the logical threads of control making up the current task.

19/5 NOTE A statement_identifier that appears immediately within the declarative region of a named loop_statement or an accept_statement is nevertheless implicitly declared immediately within the declarative region of the innermost enclosing body or block_statement; in other words, the expanded name for a named statement is not affected by whether the statement occurs inside or outside a named loop or an accept_statement — only nesting within block_statements is relevant to the form of its expanded name.

Examples

Examples of labeled statements:

```
<<Here>> <<Ici>> <<Aqui>> <<Hier>> null;
<<After>> X := 1;
```

5.2 Assignment Statements

An assignment_statement replaces the current value of a variable with the result of evaluating an expression.

Syntax

assignment_statement ::=
  *variable*_name := expression;

The execution of an assignment_statement includes the evaluation of the expression and the *assignment* of the value of the expression into the *target*. An assignment operation (as opposed to an assignment_-statement) is performed in other contexts as well, including object initialization and by-copy parameter passing. The *target* of an assignment operation is the view of the object to which a value is being assigned; the target of an assignment_statement is the variable denoted by the *variable*_name.

Name Resolution Rules

The *variable*_name of an assignment_statement is expected to be of any nonlimited type. The expected type for the expression is the type of the target.

Legality Rules

The target denoted by the *variable*_name shall be a variable of a nonlimited type.

If the target is of a tagged class-wide type *T*'Class, then the expression shall either be dynamically tagged, or of type *T* and tag-indeterminate (see 3.9.2).

Dynamic Semantics

For the execution of an assignment_statement, the *variable*_name and the expression are first evaluated in an arbitrary order.

When the type of the target is class-wide:

- If the expression is tag-indeterminate (see 3.9.2), then the controlling tag value for the expression is the tag of the target;
- Otherwise (the expression is dynamically tagged), a check is made that the tag of the value of the expression is the same as that of the target; if this check fails, Constraint_Error is raised.

The value of the expression is converted to the subtype of the target. The conversion can raise an exception (see 4.6).

In cases involving controlled types, the target is finalized, and an anonymous object can be used as an intermediate in the assignment, as described in 7.6.1, “Completion and Finalization”. In any case, the converted value of the expression is then *assigned* to the target, which consists of the following two steps:

- The value of the target becomes the converted value.
- If any part of the target is controlled, its value is adjusted as explained in 7.6.

NOTE The tag of an object never changes; in particular, an assignment_statement does not change the tag of the target.

16/2 *This paragraph was deleted.*

Examples

17 *Examples of assignment statements:*

18
```
Value := Max_Value - 1;
Shade := Blue;
```

19
```
Next_Frame(F)(M, N) := 2.5;          -- see 4.1.1
U := Dot_Product(V, W);              -- see 6.3
```

20/4
```
Writer := (Status => Open, Unit => Printer, Line_Count => 60);
                                     -- see 3.8.1
Next.all := (72074, null, Head);     -- see 3.10.1
```

21 *Examples involving scalar subtype conversions:*

22
```
I, J : Integer range 1 .. 10 := 5;
K    : Integer range 1 .. 20 := 15;
 ...
```

23
```
I := J;  -- identical ranges
K := J;  -- compatible ranges
J := K;  -- will raise Constraint_Error if K > 10
```

24 *Examples involving array subtype conversions:*

25
```
A : String(1 .. 31);
B : String(3 .. 33);
 ...
```

26
```
A := B;  -- same number of components
```

27
```
A(1 .. 9)  := "tar sauce";
A(4 .. 12) := A(1 .. 9);  -- A(1 .. 12) = "tartar sauce"
```

28/5 Assignment_statements are well-defined even in the case of overlapping slices of the same array, because the *variable*_name and expression are both evaluated before copying the value into the variable. In the above example, an implementation yielding `A(1 .. 12) = "tartartartar"` would be incorrect.

5.2.1 Target Name Symbols

1/5 @, known as the *target name* of an assignment statement, provides an abbreviation to avoid repetition of potentially long names in assignment statements.

Syntax

2/5 target_name ::= @

Name Resolution Rules

3/5 If a target_name occurs in an assignment_statement *A*, the *variable*_name *V* of *A* is a complete context. The target name is a constant view of *V*, having the nominal subtype of *V*.

Legality Rules

4/5 A target_name shall appear only in the expression of an assignment_statement.

Dynamic Semantics

5/5 For the execution of an assignment_statement with one or more target_names appearing in its expression, the *variable*_name *V* of the assignment_statement is evaluated first to determine the object denoted by *V*, and then the expression of the assignment_statement is evaluated with the evaluation of each target_name yielding a constant view of the target whose properties are otherwise identical to those

of the view provided by *V*. The remainder of the execution of the assignment_statement is as given in 5.2.

Examples

Examples of the use of target name symbols: 6/5

```
Board(1, 1) := @ + 1.0;                                                    7/5
               -- An abbreviation for Board(1, 1) := Board(1, 1) + 1.0;
               -- (Board is declared in 3.6.1).
My_Complex_Array : array (1 .. Max) of Complex; -- See 3.3.2, 3.8.          8/5
...
-- Square the element in the Count (see 3.3.1) position:
My_Complex_Array (Count) := (Re => @.Re**2 - @.Im**2,
                             Im => 2.0 * @.Re * @.Im);
    -- A target_name can be used multiple times and
    -- as a prefix if desired.
```

5.3 If Statements

An if_statement selects for execution at most one of the enclosed sequences_of_statements, depending on the (truth) value of one or more corresponding conditions. 1

Syntax

```
if_statement ::=                                                           2
   if condition then
     sequence_of_statements
   {elsif condition then
     sequence_of_statements}
   [else
     sequence_of_statements]
   end if;
```

Paragraphs 3 and 4 were deleted.

Dynamic Semantics

For the execution of an if_statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** True **then**), until one evaluates to True or all conditions are evaluated and yield False. If a condition evaluates to True, then the corresponding sequence_of_statements is executed; otherwise, none of them is executed. 5/3

Examples

Examples of if statements: 6

```
if Month = December and Day = 31 then                                      7
   Month := January;
   Day   := 1;
   Year  := Year + 1;
end if;
```

```
if Line_Too_Short then                                                     8
   raise Layout_Error;
elsif Line_Full then
   New_Line;
   Put(Item);
else
   Put(Item);
end if;
```

9
```
if My_Car.Owner.Vehicle /= My_Car then          -- see 3.10.1
   Report ("Incorrect data");
end if;
```

5.4 Case Statements

1 A case_statement selects for execution one of a number of alternative sequences_of_statements; the chosen alternative is defined by the value of an expression.

Syntax

2/3
```
case_statement ::=
  case selecting_expression is
     case_statement_alternative
    {case_statement_alternative}
  end case;
```

3
```
case_statement_alternative ::=
  when discrete_choice_list =>
    sequence_of_statements
```

Name Resolution Rules

4/3 The *selecting_*expression is expected to be of any discrete type. The expected type for each discrete_choice is the type of the *selecting_*expression.

Legality Rules

5/3 The choice_expressions, subtype_indications, and ranges given as discrete_choices of a case_statement shall be static. A discrete_choice **others**, if present, shall appear alone and in the last discrete_choice_list.

6/3 The possible values of the *selecting_*expression shall be covered (see 3.8.1) as follows:

7/4 - If the *selecting_*expression is a name (including a type_conversion, qualified_expression, or function_call) having a static and constrained nominal subtype, then each non-**others** discrete_choice shall cover only values in that subtype that satisfy its predicates (see 3.2.4), and each value of that subtype that satisfies its predicates shall be covered by some discrete_choice (either explicitly or by **others**).

8/3 - If the type of the *selecting_*expression is *root_integer*, *universal_integer*, or a descendant of a formal scalar type, then the case_statement shall have an **others** discrete_choice.

9/3 - Otherwise, each value of the base range of the type of the *selecting_*expression shall be covered (either explicitly or by **others**).

10 Two distinct discrete_choices of a case_statement shall not cover the same value.

Dynamic Semantics

11/3 For the execution of a case_statement, the *selecting_*expression is first evaluated.

12/3 If the value of the *selecting_*expression is covered by the discrete_choice_list of some case_statement_alternative, then the sequence_of_statements of the _alternative is executed.

13 Otherwise (the value is not covered by any discrete_choice_list, perhaps due to being outside the base range), Constraint_Error is raised.

NOTE The execution of a case_statement chooses one and only one alternative. Qualification of the expression of a case_statement by a static subtype can often be used to limit the number of choices that can be given explicitly. 14/5

Examples

Examples of case statements: 15

```
case Sensor is                                                          16
   when Elevation  => Record_Elevation(Sensor_Value);
   when Azimuth    => Record_Azimuth  (Sensor_Value);
   when Distance   => Record_Distance (Sensor_Value);
   when others     => null;
end case;

case Today is                                                           17
   when Mon        => Compute_Initial_Balance;
   when Fri        => Compute_Closing_Balance;
   when Tue .. Thu => Generate_Report(Today);
   when Sat .. Sun => null;
end case;

case Bin_Number(Count) is                                               18
   when 1     => Update_Bin(1);
   when 2     => Update_Bin(2);
   when 3 | 4 =>
      Empty_Bin(1);
      Empty_Bin(2);
   when others     => raise Error;
end case;
```

5.5 Loop Statements

A loop_statement includes a sequence_of_statements that is to be executed repeatedly, zero or more times with the iterations running sequentially or concurrently with one another. 1/5

Syntax

loop_statement ::= 2
  [*loop_*statement_identifier:]
    [iteration_scheme] **loop**
      sequence_of_statements
    end loop [*loop_*identifier];

iteration_scheme ::= **while** condition 3/5
  | **for** loop_parameter_specification
  | **for** iterator_specification
  | [**parallel** [aspect_specification]]
    for procedural_iterator
  | **parallel** [(chunk_specification)] [aspect_specification]
    for loop_parameter_specification
  | **parallel** [(chunk_specification)] [aspect_specification]
    for iterator_specification

chunk_specification ::= 3.1/5
    *integer_*simple_expression
  | defining_identifier **in** discrete_subtype_definition

loop_parameter_specification ::= 4/5
  defining_identifier **in** [**reverse**] discrete_subtype_definition
    [iterator_filter]

4.1/5 iterator_filter ::= **when** condition

5 If a loop_statement has a *loop*_statement_identifier, then the identifier shall be repeated after the **end loop**; otherwise, there shall not be an identifier after the **end loop**.

5.1/5 An iteration_scheme that begins with the reserved word **parallel** shall not have the reserved word **reverse** in its loop_parameter_specification.

Name Resolution Rules

5.2/5 In a chunk_specification that is an *integer*_simple_expression, the *integer*_simple_expression is expected to be of any integer type.

Static Semantics

6/5 A loop_parameter_specification declares a *loop parameter*, which is an object whose subtype (and nominal subtype) is that defined by the discrete_subtype_definition.

6.1/5 In a chunk_specification that has a discrete_subtype_definition, the chunk_specification declares a *chunk parameter* object whose subtype (and nominal subtype) is that defined by the discrete_subtype_definition.

Dynamic Semantics

6.2/5 The *filter* of an *iterator construct* (a loop_parameter_specification, iterator_specification, or procedural_iterator) is defined to be *satisfied* when there is no iterator_filter for the iterator construct, or when the condition of the iterator_filter evaluates to True for a given iteration of the iterator construct.

6.3/5 If a sequence_of_statements of a loop_statement with an iterator construct is said to be *conditionally executed*, then the statements are executed only when the filter of the iterator construct is satisfied.

6.4/5 The loop iterators loop_parameter_specification and iterator_specification can also be used in contexts other than loop_statements (for example, see 4.3.5 and 4.5.8). In such a context, the iterator *conditionally produces* values in the order specified for the associated construct below or in 5.5.2. The values produced are the values given to the loop parameter when the filter of the iterator construct is satisfied for that value. No value is produced when the condition of an iterator_filter evaluates to False.

7/5 For the execution of a loop_statement, the sequence_of_statements is executed zero or more times, until the loop_statement is complete. The loop_statement is complete when a transfer of control occurs that transfers control out of the loop, or, in the case of an iteration_scheme, as specified below.

8 For the execution of a loop_statement with a **while** iteration_scheme, the condition is evaluated before each execution of the sequence_of_statements; if the value of the condition is True, the sequence_of_statements is executed; if False, the execution of the loop_statement is complete.

8.1/5 If the reserved word **parallel** is present in the iteration_scheme of a loop_statement (a *parallel loop*), the iterations are partitioned into one or more *chunks*, each with its own separate logical thread of control (see Clause 9). If a chunk_specification is present in a parallel loop, it is elaborated first, and the result of the elaboration determines the maximum number of chunks used for the parallel loop. If the chunk_specification is an *integer*_simple_expression, the elaboration evaluates the expression, and the value of the expression determines the maximum number of chunks. If a discrete_subtype_definition is present, the elaboration elaborates the discrete_subtype_definition, which defines the subtype of the chunk parameter, and the number of values in this subtype determines the maximum number of chunks. After elaborating the chunk_specification, a check is made that the determined maximum number of chunks is greater than zero. If this check fails, Program_Error is raised.

For the execution of a loop_statement that has an iteration_scheme including a loop_parameter_specification, after elaborating the chunk_specification and aspect_specification, if any, the loop_parameter_specification is elaborated. This elaborates the discrete_subtype_definition, which defines the subtype of the loop parameter. If the discrete_subtype_definition defines a subtype with a null range, the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is conditionally executed once for each value of the discrete subtype defined by the discrete_subtype_definition that satisfies the predicates of the subtype (or until the loop is left as a consequence of a transfer of control). Prior to each such iteration, the corresponding value of the discrete subtype is assigned to the loop parameter associated with the given iteration. If the loop is a parallel loop, each chunk has its own logical thread of control with its own copy of the loop parameter; otherwise (a *sequential loop*), a single logical thread of control performs the loop, and there is a single copy of the loop parameter. Each logical thread of control handles a distinct subrange of the values of the subtype of the loop parameter such that all values are covered with no overlaps. Within each logical thread of control, the values are assigned to the loop parameter in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order. In the absence of a transfer of control, the associated parallel construct of a loop_parameter_specification is complete when all of its logical threads of control are complete. 9/5

If a chunk_specification with a discrete_subtype_definition is present, then the logical thread of control associated with a given chunk has its own copy of the chunk parameter initialized with a distinct value from the discrete subtype defined by the discrete_subtype_definition. The values of the chunk parameters are assigned such that they increase with increasing values of the ranges covered by the corresponding loop parameters. 10/5

Whether or not a chunk_specification is present in a parallel loop, the total number of iterations of the loop represents an upper bound on the number of logical threads of control devoted to the loop. 11/5

For details about the execution of a loop_statement with the iteration_scheme including an iterator_specification, see 5.5.2. For details relating to a procedural_iterator, see 5.5.3. 12/5

NOTE 1 A loop parameter declared by a loop_parameter_specification is a constant; it cannot be updated within the sequence_of_statements of the loop (see 3.3). 13/5

NOTE 2 No separate object_declaration is expected for a loop parameter, since the loop parameter is automatically declared by the loop_parameter_specification. The scope of a loop parameter extends from the loop_parameter_specification to the end of the loop_statement, and the visibility rules are such that a loop parameter is only visible within the sequence_of_statements of the loop. 14/5

NOTE 3 The discrete_subtype_definition of a for loop is elaborated just once. Use of the reserved word **reverse** does not alter the discrete subtype defined, so that the following iteration_schemes are not equivalent; the first has a null range. 15/5

```
for J in reverse 1 .. 0
for J in 0 .. 1
```
16

Examples

Example of a loop statement without an iteration scheme: 17

```
loop
   Get(Current_Character);
   exit when Current_Character = '*';
end loop;
```
18

*Example of a loop statement with a **while** iteration scheme:* 19

```
while Bid(N).Price < Cut_Off.Price loop
   Record_Bid(Bid(N).Price);
   N := N + 1;
end loop;
```
20

21 *Example of a loop statement with a **for** iteration scheme:*

22
```
for J in Buffer'Range loop      -- works even with a null range
   if Buffer(J) /= Space then
      Put(Buffer(J));
   end if;
end loop;
```

23 *Example of a loop statement with a name:*

24
```
Summation:
   while Next /= Head loop        -- see 3.10.1
      Sum  := Sum + Next.Value;
      Next := Next.Succ;
   end loop Summation;
```

25/5 *Example of a simple parallel loop:*

26/5
```
-- see 3.6
parallel
for I in Grid'Range(1) loop
   Grid(I, 1) := (for all J in Grid'Range(2) => Grid(I,J) = True);
end loop;
```

27/5 *Example of a parallel loop with a chunk specification:*

28/5
```
declare
   subtype Chunk_Number is Natural range 1 .. 8;
```

29/5
```
   Partial_Sum,
   Partial_Max : array (Chunk_Number) of Natural := (others => 0);
   Partial_Min : array (Chunk_Number) of Natural :=
                       (others => Natural'Last);
```

30/5
```
begin
   parallel (Chunk in Chunk_Number)
   for I in Grid'Range(1) loop
      declare
         True_Count : constant Natural :=
           [for J in Grid'Range(2) =>
              (if Grid (I, J) then 1 else 0)]'Reduce("+",0);
      begin
         Partial_Sum (Chunk) := @ + True_Count;
         Partial_Min (Chunk) := Natural'Min(@, True_Count);
         Partial_Max (Chunk) := Natural'Max(@, True_Count);
      end;
   end loop;
```

31/5
```
   Put_Line
     ("Total=" & Partial_Sum'Reduce("+", 0)'Image &
      ", Min=" & Partial_Min'Reduce(Natural'Min, Natural'Last)'Image &
      ", Max=" & Partial_Max'Reduce(Natural'Max, 0)'Image);
end;
```

32/5 *For an example of an* iterator_filter, *see 4.5.8.*

5.5.1 User-Defined Iterator Types

Static Semantics

1/3 The following language-defined generic library package exists:

2/5
```
generic
   type Cursor;
   with function Has_Element (Position : Cursor) return Boolean;
package Ada.Iterator_Interfaces
   with Pure, Nonblocking => False is
```

```
   type Forward_Iterator is limited interface;                                    3/3
   function First (Object : Forward_Iterator) return Cursor is abstract;
   function Next (Object : Forward_Iterator; Position : Cursor)
      return Cursor is abstract;

   type Reversible_Iterator is limited interface and Forward_Iterator;            4/3
   function Last (Object : Reversible_Iterator) return Cursor is abstract;
   function Previous (Object : Reversible_Iterator; Position : Cursor)
      return Cursor is abstract;

   type Parallel_Iterator is limited interface and Forward_Iterator;              4.1/5

   subtype Chunk_Index is Positive;                                               4.2/5

   function Is_Split (Object : Parallel_Iterator)                                 4.3/5
      return Boolean is abstract;

   procedure Split_Into_Chunks (Object     : in out Parallel_Iterator;            4.4/5
                                Max_Chunks : in     Chunk_Index) is abstract
      with Pre'Class   => not Object.Is_Split or else raise Program_Error,
           Post'Class  => Object.Is_Split and then
                          Object.Chunk_Count <= Max_Chunks;

   function Chunk_Count (Object : Parallel_Iterator)                              4.5/5
      return Chunk_Index is abstract
      with Pre'Class   => Object.Is_Split or else raise Program_Error;

   function First (Object : Parallel_Iterator;                                    4.6/5
                   Chunk  : Chunk_Index) return Cursor is abstract
      with Pre'Class   => (Object.Is_Split and then
                              Chunk <= Object.Chunk_Count)
                          or else raise Program_Error;

   function Next (Object   : Parallel_Iterator;                                   4.7/5
                  Position : Cursor;
                  Chunk    : Chunk_Index) return Cursor is abstract
      with Pre'Class   => (Object.Is_Split and then
                              Chunk <= Object.Chunk_Count)
                          or else raise Program_Error;

   type Parallel_Reversible_Iterator is limited interface                         4.8/5
      and Parallel_Iterator and Reversible_Iterator;

end Ada.Iterator_Interfaces;                                                      5/3
```

An *iterator type* is a type descended from the Forward_Iterator interface from some instance of Ada.Iterator_Interfaces. A *reversible iterator type* is a type descended from the Reversible_Iterator interface from some instance of Ada.Iterator_Interfaces. A *parallel iterator type* is a type descended from the Parallel_Iterator interface from some instance of Ada.Iterator_Interfaces. A type descended from the Parallel_Reversible_Iterator interface from some instance of Ada.Iterator_Interfaces is both a parallel iterator type and a reversible iterator type. An *iterator object* is an object of an iterator type. A *reversible iterator object* is an object of a reversible iterator type. A *parallel iterator object* is an object of a parallel iterator type. The formal subtype Cursor from the associated instance of Ada.Iterator_Interfaces is the *iteration cursor subtype* for the iterator type. 6/5

The following type-related operational aspects may be specified for an indexable container type *T* (see 4.1.6): 7/3

Default_Iterator 8/5

> This aspect is specified by a name that denotes exactly one function declared immediately within the same declaration list in which *T*, or the declaration completed by *T*, is declared, whose first parameter is of type *T* or *T*'Class or an access parameter whose designated type is type *T* or *T*'Class, whose other parameters, if any, have default expressions, and whose result type is an iterator type. This function is the *default iterator function* for *T*. Its result subtype is the *default iterator subtype* for *T*. The iteration cursor subtype for the default iterator subtype is the *default cursor subtype* for *T*. This aspect is inherited by descendants of type *T* (including *T*'Class).

9/5 Iterator_Element

This aspect is specified by a name that denotes a subtype. This is the *default element subtype* for *T*. This aspect is inherited by descendants of type *T* (including *T*'Class).

9.1/5 Iterator_View

This aspect is specified by a name that denotes a type *T2* with the following properties:

9.2/5 • *T2* is declared in the same compilation unit as *T*;

9.3/5 • *T2* is an iterable container type;

9.4/5 • *T2* has a single discriminant which is an access discriminant designating *T*; and

9.5/5 • The default iterator subtypes for *T* and *T2* statically match.

9.6/5 This aspect is never inherited, even by *T*'Class.

10/5 *This paragraph was deleted.*

11/5 An *iterable container type* is an indexable container type with specified Default_Iterator and Iterator_Element aspects. A *reversible iterable container type* is an iterable container type with the default iterator type being a reversible iterator type. A *parallel iterable container type* is an iterable container type with the default iterator type being a parallel iterator type. An *iterable container object* is an object of an iterable container type. A *reversible iterable container object* is an object of a reversible iterable container type. A *parallel iterable container object* is an object of a parallel iterable container type.

11.1/4 The Default_Iterator and Iterator_Element aspects are nonoverridable (see 13.1.1).

Legality Rules

12/3 The Constant_Indexing aspect (if any) of an iterable container type *T* shall denote exactly one function with the following properties:

13/3 • the result type of the function is covered by the default element type of *T* or is a reference type (see 4.1.5) with an access discriminant designating a type covered by the default element type of *T*;

14/3 • the type of the second parameter of the function covers the default cursor type for *T*;

15/3 • if there are more than two parameters, the additional parameters all have default expressions.

16/3 This function (if any) is the *default constant indexing function* for *T*.

17/3 The Variable_Indexing aspect (if any) of an iterable container type *T* shall denote exactly one function with the following properties:

18/3 • the result type of the function is a reference type (see 4.1.5) with an access discriminant designating a type covered by the default element type of *T*;

19/3 • the type of the second parameter of the function covers the default cursor type for *T*;

20/3 • if there are more than two parameters, the additional parameters all have default expressions.

21/3 This function (if any) is the *default variable indexing function* for *T*.

Erroneous Execution

22/5 A call on the First or Next operation on a given Parallel_Iterator object with a given Chunk value, which does not propagate an exception, should return a Cursor value that either yields False when passed to Has_Element, or that identifies an element distinct from any Cursor value returned by a call on a First or Next operation on the same Parallel_Iterator object with a different Chunk value. If the First or Next operations with a Chunk parameter behave in any other manner, execution is erroneous.

5.5.2 Generalized Loop Iteration

Generalized forms of loop iteration are provided by an iterator_specification. 1/3

Syntax

iterator_specification ::= 2/5
  defining_identifier [: loop_parameter_subtype_indication] **in** [**reverse**] *iterator*_name
    [iterator_filter]
 | defining_identifier [: loop_parameter_subtype_indication] **of** [**reverse**] *iterable*_name
    [iterator_filter]

loop_parameter_subtype_indication ::= subtype_indication | access_definition 2.1/5

If an iterator_specification is for a parallel construct, the reserved word **reverse** shall not appear in the iterator_specification. 2.2/5

Name Resolution Rules

For the first form of iterator_specification, called a *generalized iterator*, the expected type for the *iterator*_name is any iterator type. For the second form of iterator_specification, the expected type for the *iterable*_name is any array or iterable container type. If the *iterable*_name denotes an array object, the iterator_specification is called an *array component iterator*; otherwise it is called a *container element iterator*. 3/3

Legality Rules

If the reserved word **reverse** appears, the iterator_specification is a *reverse iterator*. If the iterator_specification is for a parallel construct, the iterator_specification is a *parallel iterator*. Otherwise, it is a *forward iterator*. Forward and reverse iterators are collectively called *sequential* iterators. In a reverse generalized iterator, the *iterator*_name shall be of a reversible iterator type. In a parallel generalized iterator, the *iterator*_name shall be of a parallel iterator type. In a reverse container element iterator, the default iterator type for the type of the *iterable*_name shall be a reversible iterator type. In a parallel container element iterator, the default iterator type for the type of the *iterable*_name shall be of a parallel iterator type. 4/5

The subtype defined by the loop_parameter_subtype_indication, if any, of a generalized iterator shall statically match the iteration cursor subtype. The subtype defined by the loop_parameter_subtype_-indication, if any, of an array component iterator shall statically match the component subtype of the type of the *iterable*_name. The subtype defined by the loop_parameter_subtype_indication, if any, of a container element iterator shall statically match the default element subtype for the type of the *iterable*_name. 5/5

In a container element iterator whose *iterable*_name has type *T*, if the *iterable*_name denotes a constant or the Variable_Indexing aspect is not specified for *T*, then the Constant_Indexing aspect shall be specified for *T*. 6/3

The *iterator*_name or *iterable*_name of an iterator_specification shall denote an object for which renaming is allowed (see 8.5.1). 6.1/6

A container element iterator is illegal if the call of the default iterator function that creates the loop iterator (see below) is illegal. 6.2/4

6.3/4 A generalized iterator is illegal if the iteration cursor subtype of the *iterator*_name is a limited type at the point of the generalized iterator. A container element iterator is illegal if the default cursor subtype of the type of the *iterable*_name is a limited type at the point of the container element iterator.

Static Semantics

7/5 An iterator_specification declares a *loop parameter*. In a generalized iterator, an array component iterator, or a container element iterator, if a loop_parameter_subtype_indication is present, it determines the nominal subtype of the loop parameter. In a generalized iterator, if a loop_parameter_subtype_indication is not present, the nominal subtype of the loop parameter is the iteration cursor subtype. In an array component iterator, if a loop_parameter_subtype_indication is not present, the nominal subtype of the loop parameter is the component subtype of the type of the *iterable*_name. In a container element iterator, if a loop_parameter_subtype_indication is not present, the nominal subtype of the loop parameter is the default element subtype for the type of the *iterable*_name.

8/3 In a generalized iterator, the loop parameter is a constant. In an array component iterator, the loop parameter is a constant if the *iterable*_name denotes a constant; otherwise it denotes a variable. In a container element iterator, the loop parameter is a constant if the *iterable*_name denotes a constant, or if the Variable_Indexing aspect is not specified for the type of the *iterable*_name; otherwise it is a variable.

Dynamic Semantics

9/6 For the execution of a loop_statement with an iterator_specification, after elaborating the chunk_specification, if any, the iterator_specification is elaborated. The elaboration of an iterator_specification elaborates the subtype_indication or access_definition, if any, and evaluates the *iterator*_name or the *iterable*_name.

10/6 For a sequential generalized iterator, the loop parameter is created, and the iterator object denoted by the *iterator*_name becomes the *loop iterator*. In a forward generalized iterator, the operation First of the iterator type is called on the loop iterator, to produce the initial value for the loop parameter. If the result of calling Has_Element on the initial value is False, then the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is conditionally executed and then the Next operation of the iterator type is called with the loop iterator and the current value of the loop parameter to produce the next value to be assigned to the loop parameter. This repeats until the result of calling Has_Element on the loop parameter is False, or the loop is left as a consequence of a transfer of control. For a reverse generalized iterator, the operations Last and Previous are called rather than First and Next.

10.1/6 For a parallel generalized iterator, the chunk_specification, if any, of the associated parallel construct, is used to determine the maximum number of chunks (see 5.5), and then the operation Split_Into_Chunks of the iterator type is called, with the determined maximum passed as the Max_Chunks parameter, specifying the upper bound for the number of loop parameter objects (and the number of logical threads of control) to be associated with the iterator. In the absence of a chunk_specification, the maximum number of chunks is determined in an implementation-defined manner.

10.2/5 Upon return from Split_Into_Chunks, the actual number of chunks for the loop is determined by calling the Chunk_Count operation of the iterator, at which point one logical thread of control is initiated for each chunk, with an associated chunk index in the range from one to the actual number of chunks.

10.3/5 Within each logical thread of control, a loop parameter is created. If a chunk_specification with a discrete_subtype_definition is present in the associated parallel construct, then a chunk parameter is created and initialized with a value from the discrete subtype defined by the discrete_subtype_definition, so that the order of the chosen chunk parameter values correspond to the order of the chunk indices associated with the logical threads of control. The operation First of the iterator type that has a Chunk

parameter is called on the loop iterator, with Chunk initialized from the corresponding chunk index, to produce the initial value for the loop parameter. If the result of calling Has_Element on this initial value is False, then the execution of the logical thread of control is complete. Otherwise, the sequence_of_statements is conditionally executed, and then the Next operation of the iterator type that has a Chunk parameter is called with the loop iterator, the current value of the loop parameter, and the corresponding chunk index, to produce the next value to be assigned to the loop parameter. This repeats until the result of calling Has_Element on the loop parameter is False, or the associated parallel construct is left as a consequence of a transfer of control.

In the absence of a transfer of control, the associated parallel construct of a parallel generalized iterator is complete when all of its logical threads of control are complete. 10.4/5

For an array component iterator, the chunk_specification of the associated parallel construct, if any, is used to determine the maximum number of chunks (see 5.5). The array object denoted by the *iterable*_name becomes the *array for the loop*. If the array for the loop is a null array, then the execution of the loop_statement is complete. Otherwise, the sequence_of_statements is conditionally executed with the loop parameter denoting each component of the array for the loop, using a *canonical* order of components, which is last dimension varying fastest (unless the array has convention Fortran, in which case it is first dimension varying fastest). For a forward array component iterator, the iteration starts with the component whose index values are each the first in their index range, and continues in the canonical order. For a reverse array component iterator, the iteration starts with the component whose index values are each the last in their index range, and continues in the reverse of the canonical order. For a parallel array component iterator, the iteration is broken up into contiguous chunks of the canonical order, such that all components are covered with no overlaps; each chunk has its own logical thread of control with its own loop parameter and iteration within each chunk is in the canonical order. The number of chunks is implementation defined, but is limited in the presence of a chunk_specification to the determined maximum. The loop iteration proceeds until the sequence_of_statements has been conditionally executed for each component of the array for the loop, or until the loop is left as a consequence of a transfer of control. 11/6

If a chunk_specification with a discrete_subtype_definition is present in the associated parallel construct, then the logical thread of control associated with a given chunk has a chunk parameter initialized with a distinct value from the discrete subtype defined by the discrete_subtype_definition. The values of the chunk parameters are assigned such that they increase in the canonical order of the starting array components for the chunks. 11.1/5

For a container element iterator, the chunk_specification of the associated parallel construct, if any, is used to determine the maximum number of chunks (see 5.5). If the container type has Iterator_View specified, an object of the Iterator_View type is created with the discriminant referencing the iterable container object denoted by the *iterable*_name. This is the *iterable container object for the loop*. Otherwise, the iterable container object denoted by the *iterable*_name becomes the iterable container object for the loop. The default iterator function for the type of the iterable container object for the loop is called on the iterable container object and the result is the *loop iterator*. For a sequential container element iterator, an object of the default cursor subtype is created (the *loop cursor*). For a parallel container element iterator, each chunk of iterations will have its own loop cursor, again of the default cursor subtype. 12/6

A container element iterator then proceeds as described above for a generalized iterator, except that each reference to a loop parameter is replaced by a reference to the corresponding loop cursor. For a container element iterator, the loop parameter for each iteration instead denotes an indexing (see 4.1.6) into the iterable container object for the loop, with the only parameter to the indexing being the value of the loop 13/5

cursor for the given iteration. If the loop parameter is a constant (see above), then the indexing uses the default constant indexing function for the type of the iterable container object for the loop; otherwise it uses the default variable indexing function.

14/4 Any exception propagated by the execution of a generalized iterator or container element iterator is propagated by the immediately enclosing loop statement.

Examples

15/5 *Example of a parallel generalized loop over an array:*

16/5
```
parallel
for Element of Board loop  -- See 3.6.1.
   Element := Element * 2.0; -- Double each element of Board, a two-dimensional array.
end loop;
```

17/5 *For examples of use of generalized iterators, see A.18.33 and the corresponding container packages in A.18.2 and A.18.3.*

5.5.3 Procedural Iterators

1/5 A procedural_iterator invokes a user-defined procedure, passing in the body of the enclosing loop_statement as a parameter of an anonymous access-to-procedure type, to allow the loop body to be executed repeatedly as part of the invocation of the user-defined procedure.

Syntax

2/5 procedural_iterator ::=
iterator_parameter_specification **of** iterator_procedure_call
[iterator_filter]

3/5 iterator_parameter_specification ::=
formal_part
| (defining_identifier{, defining_identifier})

4/5 iterator_procedure_call ::=
*procedure_*name
| *procedure_*prefix iterator_actual_parameter_part

5/5 iterator_actual_parameter_part ::=
(iterator_parameter_association {, iterator_parameter_association})

6/5 iterator_parameter_association ::=
parameter_association
| parameter_association_with_box

7/5 parameter_association_with_box ::=
[*formal_parameter_*selector_name =>] <>

8/5 At most one iterator_parameter_association within an iterator_actual_parameter_part shall be a parameter_association_with_box.

Name Resolution Rules

9/5 The name or prefix given in an iterator_procedure_call shall resolve to denote a callable entity *C* (the *iterating procedure*) that is a procedure, or an entry renamed as (viewed as) a procedure. When there is an

iterator_actual_parameter_part, the prefix can be an implicit_dereference of an access-to-subprogram value.

An iterator_procedure_call without a parameter_association_with_box is equivalent to one with an iterator_actual_parameter_part with an additional parameter_association_with_box at the end, with the *formal_parameter*_selector_name identifying the last formal parameter of the callable entity denoted by the name or prefix. 10/5

An iterator_procedure_call shall contain at most one iterator_parameter_association for each formal parameter of the callable entity *C*. Each formal parameter without an iterator_parameter_association shall have a default_expression (in the profile of the view of *C* denoted by the name or prefix). 11/5

The formal parameter of the callable entity *C* associated with the parameter_association_with_box shall be of an anonymous access-to-procedure type *A*. 12/5

Legality Rules

The anonymous access-to-procedure type *A* shall have at least one formal parameter in its parameter profile. If the iterator_parameter_specification is a formal_part, then this formal_part shall be mode conformant with that of *A*. If the iterator_parameter_specification is a list of defining_identifiers, the number of formal parameters of *A* shall be the same as the length of this list. 13/5

If the name or prefix given in an iterator_procedure_call denotes an abstract subprogram, the subprogram shall be a dispatching subprogram. 14/5

Static Semantics

A loop_statement with an iteration_scheme that has a procedural_iterator is equivalent to a local declaration of a procedure P followed by a procedure_call_statement that is formed from the iterator_procedure_call by replacing the <> of the parameter_association_with_box with P'Access. The formal_part of the locally declared procedure P is formed from the formal_part of the anonymous access-to-procedure type *A*, by replacing the identifier of each formal parameter of this formal_part with the identifier of the corresponding formal parameter or element of the list of defining_identifiers given in the iterator_parameter_specification. The body of *P* consists of the conditionally executed sequence_of_statements. The procedure P is called the *loop body procedure*. 15/5

In a procedural iterator, the Parallel_Calls aspect (see 9.10.1) of the loop body procedure is True if the reserved word **parallel** occurs in the corresponding loop statement, and False otherwise. 16/5

The following aspects may be specified for a callable entity *S* that has exactly one formal parameter of an anonymous access-to-subprogram type: 17/5

Allows_Exit The Allows_Exit aspect is of type Boolean. The specified value shall be static. The Allows_Exit aspect of an inherited primitive subprogram is True if Allows_Exit is True either for the corresponding subprogram of the progenitor type or for any other inherited subprogram that it overrides. If not specified or inherited as True, the Allows_Exit aspect of a callable entity is False. For an entry, only a confirming specification of False is permitted for the Allows_Exit aspect. 18/5

Specifying the Allows_Exit aspect to be True for a subprogram indicates that the subprogram *allows exit*, meaning that it is prepared to be completed by arbitrary transfers of control from the loop body procedure, including propagation of exceptions. A subprogram for which Allows_Exit is True should use finalization as appropriate rather than exception handling to recover resources and make any necessary final updates to data structures. 19/5

20/5 Parallel_Iterator

The Parallel_Iterator aspect is of type Boolean. The specified value shall be static. The Parallel_Iterator aspect of an inherited primitive subprogram is True if Parallel_Iterator is True either for the corresponding subprogram of the progenitor type or for any other inherited subprogram that it overrides. If not specified or inherited as True, the Parallel_Iterator aspect of a callable entity is False.

21/5 Specifying the Parallel_Iterator aspect to be True for a callable entity indicates that the entity is allowed to invoke the loop body procedure from multiple distinct logical threads of control. The Parallel_Iterator aspect for a subprogram shall be statically False if the subprogram allows exit.

Legality Rules

22/5 If a callable entity overrides an inherited dispatching subprogram that allows exit, the overriding callable entity also shall allow exit. If a callable entity overrides an inherited dispatching subprogram that has a True Parallel_Iterator aspect, the overriding callable entity also shall have a True Parallel_Iterator aspect.

23/5 A loop_statement with a procedural_iterator as its iteration_scheme shall begin with the reserved word **parallel** if and only if the callable entity identified in the iterator_procedure_call has a Parallel_iterator aspect of True.

24/5 If the actual parameter of an anonymous access-to-subprogram type, passed in an explicit call of a subprogram for which the Parallel_Iterator aspect is True, is of the form *P*'Access, the designated subprogram *P* shall have a Parallel_Calls aspect True (see 9.10.1).

25/5 The sequence_of_statements of a loop_statement with a procedural_iterator as its iteration_scheme shall contain an exit_statement, return statement, goto_statement, or requeue_statement that leaves the loop only if the callable entity associated with the procedural_iterator allows exit.

26/5 The sequence_of_statements of a loop_statement with a procedural_iterator as its iteration_scheme shall not contain an accept_statement whose entry_declaration occurs outside the loop_statement.

Dynamic Semantics

27/5 For the execution of a loop_statement with an iteration_scheme that has a procedural_iterator, the procedure denoted by the name or prefix of the iterator_procedure_call (the *iterating procedure*) is invoked, passing an access value designating the loop body procedure as a parameter. The iterating procedure then calls the loop body procedure zero or more times and returns, whereupon the loop_statement is complete. If the **parallel** reserved word is present, the iterating procedure is allowed to invoke the loop body procedure from multiple distinct logical threads of control. The aspect_specification, if any, is elaborated prior to the invocation of the iterating procedure.

Bounded (Run-Time) Errors

28/5 If the callable entity identified in the iterator_procedure_call allows exit, then it is a bounded error for a call of the loop body procedure to be performed from within an abort-deferred operation (see 9.8), unless the entire loop_statement was within the same abort-deferred operation. If detected, Program_Error is raised at the point of the call; otherwise, a transfer of control from the sequence_of_statements of the loop_statement will not necessarily terminate the loop_statement, and the loop body procedure can be called again.

29/5 If a loop_statement with the procedural_iterator as its iteration_scheme (see 5.5) does not begin with the reserved word **parallel**, it is a bounded error if the loop body procedure is invoked from a different logical thread of control than the one that initiates the loop_statement. If detected, Program_Error is raised;

otherwise, conflicts associated with concurrent executions of the loop body procedure can occur without being detected by the applicable conflict check policy (see 9.10.1). Furthermore, propagating an exception or making an attempt to exit in the presence of multiple threads of control will not necessarily terminate the loop_statement, deadlock can occur, or the loop body procedure can be called again.

Examples

Example of iterating over a map from My_Key_Type to My_Element_Type (see A.18.4): 30/5

```
for (C : Cursor) of My_Map.Iterate loop                                    31/5
   Put_Line (My_Key_Type'Image (Key (C)) & " => " &
      My_Element_Type'Image (Element (C)));
end loop;
```

-- The above is equivalent to: 32/5

```
declare                                                                    33/5
   procedure P (C : Cursor) is
   begin
      Put_Line (My_Key_Type'Image (Key (c)) & " => " &
         My_Element_Type'Image (Element (C)));
   end P;
begin
   My_Map.Iterate (P'Access);
end;
```

Example of iterating over the environment variables (see A.17): 34/5

```
for (Name, Val) of Ada.Environment_Variables.Iterate(<>) loop              35/5
   -- "(<>)" is optional because it is the last parameter
   Put_Line (Name & " => " & Val);
end loop;
```

-- The above is equivalent to: 36/5

```
declare                                                                    37/5
   procedure P (Name : String; Val : String) is
   begin
      Put_Line (Name & " => " & Val);
   end P;
begin
   Ada.Environment_Variables.Iterate (P'Access);
end;
```

5.6 Block Statements

1 A block_statement encloses a handled_sequence_of_statements optionally preceded by a declarative_part.

Syntax

2
```
block_statement ::=
  [block_statement_identifier:]
     [declare
        declarative_part]
      begin
        handled_sequence_of_statements
      end [block_identifier];
```

3 If a block_statement has a *block_*statement_identifier, then the identifier shall be repeated after the **end**; otherwise, there shall not be an identifier after the **end**.

Static Semantics

4 A block_statement that has no explicit declarative_part has an implicit empty declarative_part.

Dynamic Semantics

5 The execution of a block_statement consists of the elaboration of its declarative_part followed by the execution of its handled_sequence_of_statements.

Examples

6 *Example of a block statement with a local variable:*

7
```
Swap:
   declare
      Temp : Integer;
   begin
      Temp := V; V := U; U := Temp;
   end Swap;
```

5.6.1 Parallel Block Statements

1/5 A parallel_block_statement comprises two or more sequence_of_statements separated by **and** where each represents an independent activity that is intended to proceed concurrently with the others.

Syntax

2/5
```
parallel_block_statement ::=
  parallel [(chunk_specification)] [aspect_specification] do
    sequence_of_statements
  and
    sequence_of_statements
  {and
    sequence_of_statements}
  end do;
```

3/5 The chunk_specification, if any, of a parallel_block_statement shall be an *integer_*simple_-expression.

Dynamic Semantics

For the execution of a parallel_block_statement, the chunk_specification and the aspect_specification, if any, are elaborated in an arbitrary order. After elaborating the chunk_specification, if any, a check is made that the determined maximum number of chunks is greater than zero. If this check fails, Program_Error is raised. 4/5

Then, the various sequence_of_statements are grouped into one or more *chunks*, each with its own logical thread of control (see Clause 9), up to the maximum number of chunks specified by the chunk_specification, if any. Within each chunk every sequence_of_statements of the chunk is executed in turn, in an arbitrary order. The parallel_block_statement is complete once every one of the sequence_of_statements has completed, either by reaching the end of its execution, or due to a transfer of control out of the construct by one of the sequence_of_statements (see 5.1). 5/5

Examples

Example of a parallel block used to walk a binary tree in parallel: 6/5

```
procedure Traverse (T : Expr_Ptr) is -- see 3.9.1
begin
   if T /= null and then
      T.all in Binary_Operation'Class -- see 3.9.1
   then -- recurse down the binary tree
      parallel do
         Traverse (T.Left);
      and
         Traverse (T.Right);
      and
         Ada.Text_IO.Put_Line
            ("Processing " & Ada.Tags.Expanded_Name (T'Tag));
      end do;
   end if;
end Traverse;
```

7/5

Example of a parallel block used to search two halves of a string in parallel: 8/5

```
function Search (S : String; Char : Character) return Boolean is
begin
   if S'Length <= 1000 then
       -- Sequential scan
       return (for some C of S => C = Char);
   else
       -- Parallel divide and conquer
       declare
          Mid : constant Positive := S'First + S'Length/2 - 1;
       begin
          parallel do
             for C of S(S'First .. Mid) loop
                if C = Char then
                   return True;  -- Terminates enclosing do
                end if;
             end loop;
          and
             for C of S(Mid + 1 .. S'Last) loop
                if C = Char then
                   return True;  -- Terminates enclosing do
                end if;
             end loop;
          end do;
          -- Not found
          return False;
       end;
   end if;
end Search;
```

9/5

5.7 Exit Statements

1 An exit_statement is used to complete the execution of an enclosing loop_statement; the completion is conditional if the exit_statement includes a condition.

Syntax

2 exit_statement ::=
exit [*loop_*name] [**when** condition];

Name Resolution Rules

3 The *loop_*name, if any, in an exit_statement shall resolve to denote a loop_statement.

Legality Rules

4 Each exit_statement *applies to* a loop_statement; this is the loop_statement being exited. An exit_statement with a name is only allowed within the loop_statement denoted by the name, and applies to that loop_statement. An exit_statement without a name is only allowed within a loop_statement, and applies to the innermost enclosing one. An exit_statement that applies to a given loop_statement shall not appear within a body or accept_statement, if this construct is itself enclosed by the given loop_statement.

Dynamic Semantics

5 For the execution of an exit_statement, the condition, if present, is first evaluated. If the value of the condition is True, or if there is no condition, a transfer of control is done to complete the loop_statement. If the value of the condition is False, no transfer of control takes place.

6 NOTE Several nested loops can be exited by an exit_statement that names the outer loop.

Examples

7 *Examples of loops with exit statements:*

8
```
for N in 1 .. Max_Num_Items loop
   Get_New_Item(New_Item);
   Merge_Item(New_Item, Storage_File);
   exit when New_Item = Terminal_Item;
end loop;
```

9
```
Main_Cycle:
   loop
      -- initial statements
      exit Main_Cycle when Found;
      -- final statements
   end loop Main_Cycle;
```

5.8 Goto Statements

1 A goto_statement specifies an explicit transfer of control from this statement to a target statement with a given label.

Syntax

2 goto_statement ::= **goto** *label_*name;

Name Resolution Rules

The *label_*name shall resolve to denote a label; the statement with that label is the *target statement*. 3

Legality Rules

The innermost sequence_of_statements that encloses the target statement shall also enclose the goto_statement. Furthermore, if a goto_statement is enclosed by an accept_statement or a body, then the target statement shall not be outside this enclosing construct. 4

Dynamic Semantics

The execution of a goto_statement transfers control to the target statement, completing the execution of any compound_statement that encloses the goto_statement but does not enclose the target. 5

NOTE The above rules allow transfer of control to a statement of an enclosing sequence_of_statements but not the reverse. Similarly, they prohibit transfers of control such as between alternatives of a case_statement, if_statement, or select_statement; between exception_handlers; or from an exception_handler of a handled_sequence_of_statements back to its sequence_of_statements. 6

Examples

Example of a loop containing a goto statement: 7

```
<<Sort>>                                                    8
for I in 1 .. N-1 loop
   if A(I) > A(I+1) then
      Exchange(A(I), A(I+1));
      goto Sort;
   end if;
end loop;
```

6 Subprograms

A subprogram is a program unit or intrinsic operation whose execution is invoked by a subprogram call. There are two forms of subprogram: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. The definition of a subprogram can be given in two parts: a subprogram declaration defining its interface, and a subprogram_body defining its execution. Operators and enumeration literals are functions. 1

A *callable entity* is a subprogram or entry (see Section 9). A callable entity is invoked by a *call*; that is, a subprogram call or entry call. A *callable construct* is a construct that defines the action of a call upon a callable entity: a subprogram_body, entry_body, or accept_statement. 2/3

6.1 Subprogram Declarations

A subprogram_declaration declares a procedure or function. 1

Syntax

subprogram_declaration ::= 2/3
   [overriding_indicator]
   subprogram_specification
      [aspect_specification];

This paragraph was deleted. 3/2

subprogram_specification ::= 4/2
   procedure_specification
 | function_specification

procedure_specification ::= 4.1/2
   procedure defining_program_unit_name parameter_profile

function_specification ::= 4.2/2
   function defining_designator parameter_and_result_profile

designator ::= [parent_unit_name .]identifier | operator_symbol 5

defining_designator ::= 6
   defining_program_unit_name | defining_operator_symbol

defining_program_unit_name ::= [parent_unit_name .]defining_identifier 7

The optional parent_unit_name is only allowed for library units (see 10.1.1). 8

operator_symbol ::= string_literal 9

The sequence of characters in an operator_symbol shall form a reserved word, a delimiter, or compound delimiter that corresponds to an operator belonging to one of the six categories of operators defined in 4.5. 10/5

defining_operator_symbol ::= operator_symbol 11

parameter_profile ::= [formal_part] 12

13/2 parameter_and_result_profile ::=
[formal_part] **return** [null_exclusion] subtype_mark
| [formal_part] **return** access_definition

14 formal_part ::=
(parameter_specification {; parameter_specification})

15/5 parameter_specification ::=
defining_identifier_list : [**aliased**] mode [null_exclusion] subtype_mark [:= default_expression]
[aspect_specification]
| defining_identifier_list : access_definition [:= default_expression]
[aspect_specification]

16 mode ::= [**in**] | **in out** | **out**

Name Resolution Rules

17 A *formal parameter* is an object directly visible within a subprogram_body that represents the actual parameter passed to the subprogram in a call; it is declared by a parameter_specification. For a formal parameter, the expected type for its default_expression, if any, is that of the formal parameter.

Legality Rules

18/3 The *parameter mode* of a formal parameter conveys the direction of information transfer with the actual parameter: **in**, **in out**, or **out**. Mode **in** is the default, and is the mode of a parameter defined by an access_definition.

19 A default_expression is only allowed in a parameter_specification for a formal parameter of mode **in**.

20/3 A subprogram_declaration or a generic_subprogram_declaration requires a completion unless the Import aspect (see B.1) is True for the declaration; the completion shall be a body or a renaming_declaration (see 8.5). A completion is not allowed for an abstract_subprogram_declaration (see 3.9.3), a null_procedure_declaration (see 6.7), or an expression_function_declaration (see 6.8).

21 A name that denotes a formal parameter is not allowed within the formal_part in which it is declared, nor within the formal_part of a corresponding body or accept_statement.

Static Semantics

22 The *profile* of (a view of) a callable entity is either a parameter_profile or parameter_and_result_profile; it embodies information about the interface to that entity — for example, the profile includes information about parameters passed to the callable entity. All callable entities have a profile — enumeration literals, other subprograms, and entries. An access-to-subprogram type has a designated profile. Associated with a profile is a calling convention. A subprogram_declaration declares a procedure or a function, as indicated by the initial reserved word, with name and profile as given by its specification.

23/2 The nominal subtype of a formal parameter is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_specification. The nominal subtype of a function result is the subtype determined by the optional null_exclusion and the subtype_mark, or defined by the access_definition, in the parameter_and_result_profile.

23.1/3 An *explicitly aliased parameter* is a formal parameter whose parameter_specification includes the reserved word **aliased**.

24/2 An *access parameter* is a formal **in** parameter specified by an access_definition. An *access result type* is a function result type specified by an access_definition. An access parameter or result type is of an

anonymous access type (see 3.10). Access parameters of an access-to-object type allow dispatching calls to be controlled by access values. Access parameters of an access-to-subprogram type permit calls to subprograms passed as parameters irrespective of their accessibility level.

The *subtypes of a profile* are: 25

- For any non-access parameters, the nominal subtype of the parameter. 26
- For any access parameters of an access-to-object type, the designated subtype of the parameter type. 27/2
- For any access parameters of an access-to-subprogram type, the subtypes of the designated profile of the parameter type. 27.1/3
- For any non-access result, the nominal subtype of the function result. 28/2
- For any access result type of an access-to-object type, the designated subtype of the result type. 28.1/2
- For any access result type of an access-to-subprogram type, the subtypes of the designated profile of the result type. 28.2/3

The *types of a profile* are the types of those subtypes. 29

A subprogram declared by an abstract_subprogram_declaration is abstract; a subprogram declared by a subprogram_declaration is not. See 3.9.3, “Abstract Types and Subprograms”. Similarly, a procedure declared by a null_procedure_declaration is a null procedure; a procedure declared by a subprogram_declaration is not. See 6.7, “Null Procedures”. Finally, a function declared by an expression_function_declaration is an expression function; a function declared by a subprogram_declaration is not. See 6.8, “Expression Functions”. 30/3

An overriding_indicator is used to indicate whether overriding is intended. See 8.3.1, “Overriding Indicators”. 30.1/2

Dynamic Semantics

The elaboration of a subprogram_declaration has no effect. 31/2

NOTE 1 A parameter_specification with several identifiers is equivalent to a sequence of single parameter_specifications, as explained in 3.3. 32

NOTE 2 Abstract subprograms do not have bodies, and cannot be used in a nondispatching call (see 3.9.3, “Abstract Types and Subprograms”). 33

NOTE 3 The evaluation of default_expressions is caused by certain calls, as described in 6.4.1. They are not evaluated during the elaboration of the subprogram declaration. 34

NOTE 4 Subprograms can be called recursively and can be called concurrently from multiple tasks. 35

Examples

Examples of subprogram declarations: 36

```
procedure Traverse_Tree;                                                    37
procedure Increment(X : in out Integer);
procedure Right_Indent(Margin : out Line_Size);          -- see 3.5.4
procedure Switch(From, To : in out Link);                -- see 3.10.1

function Random return Probability;                      -- see 3.5.7       38

function Min_Cell(X : Link) return Cell;                 -- see 3.10.1      39/4
function Next_Frame(K : Positive) return Frame;          -- see 3.10
function Dot_Product(Left, Right : Vector) return Real;  -- see 3.6
function Find(B : aliased in out Barrel; Key : String) return Real;
                                                         -- see 4.1.5

function "*"(Left, Right : Matrix) return Matrix;        -- see 3.6         40
```

41 *Examples of **in** parameters with default expressions:*

42
```
procedure Print_Header(Pages  : in Natural;
                       Header : in Line    :=  (1 .. Line'Last => ' '); -- see 3.6
                       Center : in Boolean := True);
```

6.1.1 Preconditions and Postconditions

1/5 For a noninstance subprogram (including a generic formal subprogram), a generic subprogram, an entry, or an access-to-subprogram type, the following language-defined assertion aspects may be specified with an aspect_specification (see 13.1.1):

2/5 Pre — This aspect specifies a specific precondition for a callable entity or an access-to-subprogram type; it shall be specified by an expression, called a *specific precondition expression*. If not specified for an entity, the specific precondition expression for the entity is the enumeration literal True.

3/5 Pre'Class — This aspect specifies a class-wide precondition for a dispatching operation of a tagged type and its descendants; it shall be specified by an expression, called a *class-wide precondition expression*. If not specified for an entity, then if no other class-wide precondition applies to the entity, the class-wide precondition expression for the entity is the enumeration literal True.

4/5 Post — This aspect specifies a specific postcondition for a callable entity or an access-to-subprogram type; it shall be specified by an expression, called a *specific postcondition expression*. If not specified for an entity, the specific postcondition expression for the entity is the enumeration literal True.

5/5 Post'Class — This aspect specifies a class-wide postcondition for a dispatching operation of a tagged type and its descendants; it shall be specified by an expression, called a *class-wide postcondition expression*. If not specified for an entity, the class-wide postcondition expression for the entity is the enumeration literal True.

Name Resolution Rules

6/3 The expected type for a precondition or postcondition expression is any boolean type.

7/5 Within the expression for a Pre'Class or Post'Class aspect for a primitive subprogram *S* of a tagged type *T*, a name that denotes a formal parameter (or *S*'Result) of type *T* is interpreted as though it had a (notional) nonabstract type *NT* that is a formal derived type whose ancestor type is *T*, with directly visible primitive operations. Similarly, a name that denotes a formal access parameter (or *S*'Result for an access result) of type access-to-*T* is interpreted as having type access-to-*NT*. The result of this interpretation is that the only operations that can be applied to such names are those defined for such a formal derived type.

8/5 For an attribute_reference with attribute_designator Old, if the attribute reference has an expected type (or class of types) or shall resolve to a given type, the same applies to the prefix; otherwise, the prefix shall be resolved independently of context.

Legality Rules

9/3 The Pre or Post aspect shall not be specified for an abstract subprogram or a null procedure. Only the Pre'Class and Post'Class aspects may be specified for such a subprogram.

10/3 If a type *T* has an implicitly declared subprogram *P* inherited from a parent type *T1* and a homograph (see 8.3) of *P* from a progenitor type *T2*, and

11/3 - the corresponding primitive subprogram *P1* of type *T1* is neither null nor abstract; and

12/3 - the class-wide precondition expression True does not apply to *P1* (implicitly or explicitly); and

- there is a class-wide precondition expression that applies to the corresponding primitive subprogram *P2* of *T2* that does not fully conform to any class-wide precondition expression that applies to *P1*, 13/3

then: 14/3

- If the type *T* is abstract, the implicitly declared subprogram *P* is *abstract*. 15/3
- Otherwise, the subprogram *P* *requires overriding* and shall be overridden with a nonabstract subprogram. 16/3

If a renaming of a subprogram or entry *S1* overrides an inherited subprogram *S2*, then the overriding is illegal unless each class-wide precondition expression that applies to *S1* fully conforms to some class-wide precondition expression that applies to *S2* and each class-wide precondition expression that applies to *S2* fully conforms to some class-wide precondition expression that applies to *S1*. 17/3

Pre'Class shall not be specified for an overriding primitive subprogram of a tagged type *T* unless the Pre'Class aspect is specified for the corresponding primitive subprogram of some ancestor of *T*. 17.1/4

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 17.2/4

Static Semantics

If a Pre'Class or Post'Class aspect is specified for a primitive subprogram *S* of a tagged type *T*, or such an aspect defaults to True, then a corresponding expression also applies to the corresponding primitive subprogram *S* of each descendant of *T* (including *T* itself). The *corresponding expression* is constructed from the associated expression as follows: 18/5

- References to formal parameters of *S* (or to *S* itself) are replaced with references to the corresponding formal parameters of the corresponding inherited or overriding subprogram *S* (or to the corresponding subprogram *S* itself). 18.1/4

If the primitive subprogram *S* is not abstract, but the given descendant of *T* is abstract, then a nondispatching call on *S* is illegal if the corresponding expression for any Pre'Class or Post'Class aspect that applies to *S* includes a nondispatching invocation of an abstract function. Similarly, a primitive subprogram of an abstract type *T*, to which such a Pre'Class or Post'Class aspect applies, shall neither be the prefix of an Access attribute_reference, nor shall it be a generic actual subprogram for a formal subprogram declared by a formal_concrete_subprogram_declaration. 18.2/6

If performing checks is required by the Pre, Pre'Class, Post, or Post'Class assertion policies (see 11.4.2) in effect at the point of a corresponding aspect specification applicable to a given subprogram, entry, or access-to-subprogram type, then the respective precondition or postcondition expressions are considered *enabled*. 19/5

A subexpression of a postcondition expression is *known on entry* if it is any of: 20/5

- a static subexpression (see 4.9); 21/5
- a literal whose type does not have any Integer_Literal, Real_Literal, or String_Literal aspect specified, or the function specified by such an attribute has aspect Global specified to be **null**; 22/5
- a name statically denoting a full constant declaration which is known to have no variable views (see 3.3); 22.1/5
- a name statically denoting a nonaliased **in** parameter of an elementary type; 22.2/5
- an Old attribute_reference; 22.3/5
- an invocation of a predefined operator where all of the operands are known on entry; 22.4/5

22.5/5 • a function call where the function has aspect Global => **null** where all of the actual parameters are known on entry;

22.6/5 • a selected_component of a known-on-entry prefix;

22.7/5 • an indexed_component of a known-on-entry prefix where all index expressions are known on entry;

22.8/5 • a parenthesized known-on-entry expression;

22.9/5 • a qualified_expression or type_conversion whose operand is a known-on-entry expression;

22.10/5 • a conditional_expression where all of the conditions, *selecting_*expressions, and *dependent_*expressions are known on entry.

22.11/5 A subexpression of a postcondition expression is *unconditionally evaluated*, *conditionally evaluated*, or *repeatedly evaluated*. A subexpression is considered unconditionally evaluated unless it is conditionally evaluated or repeatedly evaluated.

22.12/6 Any subexpression of an expression that appears in an iterative context (see 4.4) is *repeatedly evaluated*.

Paragraphs 22.13 through 22.15 were deleted.

22.13/6 •

22.14/6 •

22.15/6 •

22.16/5 For a subexpression that is conditionally evaluated, there is a set of *determining expressions* that determine whether the subexpression is actually evaluated at run time. Subexpressions that are conditionally evaluated and their determining expressions are as follows:

22.17/5 • For an if_expression that is not repeatedly evaluated, a subexpression of any part other than the first condition is conditionally evaluated, and its determining expressions include all conditions of the if_expression that precede the subexpression textually;

22.18/5 • For a case_expression that is not repeatedly evaluated, a subexpression of any *dependent_*expression is conditionally evaluated, and its determining expressions include the *selecting_*expression of the case_expression;

23/5 • For a short-circuit control form that is not repeatedly evaluated, a subexpression of the right-hand operand is conditionally evaluated, and its determining expressions include the left-hand operand of the short-circuit control form;

24/5 • For a membership test that is not repeatedly evaluated, a subexpression of a membership_choice other than the first is conditionally evaluated, and its determining expressions include the *tested_*simple_expression and the preceding membership_choices of the membership test.

24.1/5 A conditionally evaluated subexpression is *determined to be unevaluated* at run time if its set of determining expressions are all known on entry, and when evaluated on entry their values are such that the given subexpression is not evaluated.

25/3 For a prefix X that denotes an object of a nonlimited type, the following attribute is defined:

26/5 X'Old Each X'Old in a postcondition expression that is enabled, other than those that occur in subexpressions that are determined to be unevaluated, denotes a constant that is implicitly declared at the beginning of the subprogram body, entry body, or accept statement.

26.1/4 The implicitly declared entity denoted by each occurrence of X'Old is declared as follows:

26.2/4 • If X is of an anonymous access type defined by an access_definition *A* then

```
X'Old : constant A := X;
```
26.3/4

- If X is of a specific tagged type *T* then 26.4/4

```
anonymous : constant T'Class := T'Class(X);
X'Old : T renames T(anonymous);
```
26.5/4

  where the name X'Old denotes the object renaming. 26.6/4

- Otherwise 26.7/4

```
X'Old : constant S := X;
```
26.8/4

  where *S* is the nominal subtype of X. This includes the case where the type of *S* is an anonymous array type or a universal type. 26.9/4

The type and nominal subtype of X'Old are as implied by the above definitions. 26.10/5

Reference to this attribute is only allowed within a postcondition expression. The prefix of an Old attribute_reference shall not contain a Result attribute_reference, nor an Old attribute_reference, nor a use of an entity declared within the postcondition expression but not within prefix itself (for example, the loop parameter of an enclosing quantified_expression). The prefix of an Old attribute_reference shall statically name (see 4.9) an entity, unless the attribute_reference is unconditionally evaluated, or is conditionally evaluated where all of the determining expressions are known on entry. 27/5

For a prefix F that denotes a function declaration or an access-to-function type, the following attribute is defined: 28/5

F'Result — Within a postcondition expression for F, denotes the return object of the function call for which the postcondition expression is evaluated. The type of this attribute is that of the result subtype of the function or access-to-function type except within a Post'Class postcondition expression for a function with a controlling result or with a controlling access result; in those cases the type of the attribute is described above as part of the Name Resolution Rules for Post'Class. 29/5

Use of this attribute is allowed only within a postcondition expression for F. 30/3

For a prefix E that denotes an entry declaration of an entry family (see 9.5.2), the following attribute is defined: 30.1/5

E'Index — Within a precondition or postcondition expression for entry family E, denotes the value of the entry index for the call of E. The nominal subtype of this attribute is the entry index subtype. 30.2/5

Use of this attribute is allowed only within a precondition or postcondition expression for E. 30.3/5

Dynamic Semantics

Upon a call of the subprogram or entry, after evaluating any actual parameters, precondition checks are performed as follows: 31/3

- The specific precondition check begins with the evaluation of the specific precondition expression that applies to the subprogram or entry, if it is enabled; if the expression evaluates to False, Assertions.Assertion_Error is raised; if the expression is not enabled, the check succeeds. 32/3
- The class-wide precondition check begins with the evaluation of any enabled class-wide precondition expressions that apply to the subprogram or entry. If and only if all the class-wide precondition expressions evaluate to False, Assertions.Assertion_Error is raised. 33/3

The precondition checks are performed in an arbitrary order, and if any of the class-wide precondition expressions evaluate to True, it is not specified whether the other class-wide precondition expressions are evaluated. The precondition checks and any check for elaboration of the subprogram body are performed 34/5

in an arbitrary order. In a call on a protected operation, the checks are performed before starting the protected action. For an entry call, the checks are performed prior to checking whether the entry is open.

35/3 Upon successful return from a call of the subprogram or entry, prior to copying back any by-copy **in out** or **out** parameters, the postcondition check is performed. This consists of the evaluation of any enabled specific and class-wide postcondition expressions that apply to the subprogram or entry. If any of the postcondition expressions evaluate to False, then Assertions.Assertion_Error is raised. The postcondition expressions are evaluated in an arbitrary order, and if any postcondition expression evaluates to False, it is not specified whether any other postcondition expressions are evaluated. The postcondition check, and any constraint or predicate checks associated with **in out** or **out** parameters are performed in an arbitrary order.

35.1/4 For a call to a task entry, the postcondition check is performed before the end of the rendezvous; for a call to a protected operation, the postcondition check is performed before the end of the protected action of the call. The postcondition check for any call is performed before the finalization of any implicitly-declared constants associated (as described above) with Old attribute_references but after the finalization of any other entities whose accessibility level is that of the execution of the callable construct.

36/3 If a precondition or postcondition check fails, the exception is raised at the point of the call; the exception cannot be handled inside the called subprogram or entry. Similarly, any exception raised by the evaluation of a precondition or postcondition expression is raised at the point of call.

37/4 For any call to a subprogram or entry *S* (including dispatching calls), the checks that are performed to verify specific precondition expressions and specific and class-wide postcondition expressions are determined by those for the subprogram or entry actually invoked. Note that the class-wide postcondition expressions verified by the postcondition check that is part of a call on a primitive subprogram of type *T* includes all class-wide postcondition expressions originating in any progenitor of *T*, even if the primitive subprogram called is inherited from a type *T1* and some of the postcondition expressions do not apply to the corresponding primitive subprogram of *T1*. Any operations within a class-wide postcondition expression that were resolved as primitive operations of the (notional) formal derived type *NT*, are in the evaluation of the postcondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*. This applies to both dispatching and non-dispatching calls on *S*.

38/4 The class-wide precondition check for a call to a subprogram or entry *S* consists solely of checking the class-wide precondition expressions that apply to the denoted callable entity (not necessarily to the one that is invoked). Any operations within such an expression that were resolved as primitive operations of the (notional) formal derived type *NT* are in the evaluation of the precondition bound to the corresponding operations of the type identified by the controlling tag of the call on *S*. This applies to both dispatching and non-dispatching calls on *S*.

39/5 For the purposes of the above rules, a call on an inherited subprogram is considered to involve a call on a subprogram *S'* whose body consists only of a call (with appropriate conversions) on the non-inherited subprogram *S* from which the inherited subprogram was derived. It is not specified whether class-wide precondition or postcondition expressions that are equivalent (with respect to which non-inherited function bodies are executed) for *S* and *S'* are evaluated once or twice. If evaluated only once, the value returned is used for both associated checks.

40/5 For a call via an access-to-subprogram value, precondition and postcondition checks performed are as determined by the subprogram or entry denoted by the prefix of the Access attribute reference that produced the value. In addition, a precondition check of any precondition expression associated with the access-to-subprogram type is performed. Similarly, a postcondition check of any postcondition expression associated with the access-to-subprogram type is performed.

For a call on a generic formal subprogram, precondition and postcondition checks performed are as determined by the subprogram or entry denoted by the actual subprogram, along with any specific precondition and specific postcondition of the formal subprogram itself. 41/5

Implementation Permissions

An implementation may evaluate a known-on-entry subexpression of a postcondition expression of an entity at the place where X'Old constants are created for the entity, with the normal evaluation of the postcondition expression, or both. 42/5

NOTE 1 A precondition is checked just before the call. If another task can change any value that the precondition expression depends on, the precondition can evaluate to False within the subprogram or entry body. 43/5

NOTE 2 For an example of the use of these aspects and attributes, see the Streams Subsystem definitions in 13.13.1. 44/5

6.1.2 The Global and Global'Class Aspects

The Global and Global'Class aspects of a program unit are used to identify the objects global to the unit that can be read or written during its execution. 1/5

Syntax

global_aspect_definition ::= 2/5
  null
 | Unspecified
 | global_mode global_designator
 | (global_aspect_element{; global_aspect_element})

global_aspect_element ::= 3/5
  global_mode global_set
 | global_mode **all**
 | global_mode **synchronized**

global_mode ::= 4/5
  basic_global_mode
 | extended_global_mode

basic_global_mode ::= **in** | **in out** | **out** 5/5

global_set ::= global_name {, global_name} 6/5

global_designator ::= **all** | **synchronized** | global_name 7/5

global_name ::= *object_*name | *package_*name 8/5

Name Resolution Rules

A global_name shall resolve to statically name an object or a package (including a limited view of a package). 9/5

Static Semantics

For a subprogram, an entry, an access-to-subprogram type, a task unit, a protected unit, or a library package or generic library package, the following language-defined aspect may be specified with an aspect_specification (see 13.1.1): 10/5

Global The Global aspect shall be specified with a global_aspect_definition. 11/5

12/5 The Global aspect identifies the set of variables (which, for the purposes of this clause, includes all constants except those which are known to have no variable views (see 3.3)) that are global to a callable entity or task body, and that are read or updated as part of the execution of the callable entity or task body. If specified for a protected unit, it refers to all of the protected operations of the protected unit. Constants of any type may also be mentioned in a Global aspect.

13/5 If not specified or otherwise defined below, the aspect defaults to the Global aspect for the enclosing library unit if the entity is declared at library level, and to Unspecified otherwise. If not specified for a library unit, the aspect defaults to `Global => null` for a library unit that is declared Pure, and to `Global => Unspecified` otherwise.

14/5 For a dispatching subprogram, the following language-defined aspect may be specified with an aspect_specification (see 13.1.1):

15/5 Global'Class The Global'Class aspect shall be specified with a global_aspect_definition. This aspect identifies an upper bound on the set of variables global to a dispatching operation that can be read or updated as a result of a dispatching call on the operation. If not specified, the aspect defaults to the Global aspect for the dispatching subprogram.

16/5 Together, we refer to the Global and Global'Class aspects as *global* aspects.

17/5 A global_aspect_definition defines the Global or Global'Class aspect of some entity. The Global aspect identifies the sets of global variables that can be read, written, or modified as a side effect of executing the operation(s) associated with the entity. The Global'Class aspect associated with a dispatching operation of type *T* represents a restriction on the Global aspect on a corresponding operation of any descendant of type *T*.

18/5 The Global aspect for a callable entity defines the global variables that can be referenced as part of a call on the entity, including any assertion expressions that apply to the call (even if not enabled), such as preconditions, postconditions, predicates, and type invariants.

19/5 The Global aspect for an access-to-subprogram object (or subtype) identifies the global variables that can be referenced when calling via the object (or any object of that subtype) including assertion expressions that apply.

20/5 For a predefined operator of an elementary type, the function representing an enumeration literal, or any other static function (see 4.9), the Global aspect is **null**. For a predefined operator of a composite type, the Global aspect of the operator defaults to that of the enclosing library unit (unless a Global aspect is specified for the type — see H.7).

21/5 The following is defined in terms of operations that are performed by or on behalf of an entity. The rules on operations apply to the entity(s) associated with those operations.

22/5 The global variables associated with any global_mode can be read as a side effect of an operation. The **in out** and **out** global_modes together identify the set of global variables that can be updated as a side effect of an operation. Creating an access-to-variable value that designates an object is considered an update of the designated object, and creating an access-to-constant value that designates an object is considered a read of the designated object.

23/5 The overall set of objects associated with each global_mode includes all objects identified for the mode in the global_aspect_definition.

24/5 A global_set identifies a *global variable set* as follows:

25/5 - **all** identifies the set of all global variables;

- **synchronized** identifies the set of all synchronized variables (see 9.10), as well as variables of a composite type all of whose non-discriminant subcomponents are synchronized; 26/5
- global_name{, global_name} identifies the union of the sets of variables identified by the global_names in the list, for the following forms of global_name: 27/5
  - *object*_name identifies the specified global variable (or constant); 28/5
  - *package*_name identifies the set of all variables declared in the private part or body of the package, or anywhere within a private descendant of the package. 29/5

Legality Rules

Within a global_aspect_definition, a given global_mode shall be specified at most once. Similarly, within a global_aspect_definition, a given entity shall be named at most once by a global_name. 30/5

If an entity (other than a library package or generic library package) has a Global aspect other than Unspecified or **in out all**, then the associated operation(s) shall read only those variables global to the entity that are within the global variable set associated with the **in**, **in out**, or **out** global_modes, and the operation(s) shall update only those variables global to the entity that are within the global variable set associated with either the **in out** or **out** global_modes. In the absence of the No_Hidden_Indirect_Globals restriction (see H.4), this ignores objects reached via a dereference of an access value. The above rule includes any possible Global effects of calls occurring during the execution of the operation, except for the following excluded calls: 31/5

- calls to formal subprograms; 32/5
- calls associated with operations on formal subtypes; 33/5
- calls through formal objects of an access-to-subprogram type; 34/5
- calls through access-to-subprogram parameters; 35/5
- calls on operations with Global aspect Unspecified. 36/5

The possible Global effects of these excluded calls (other than those that are Unspecified) are taken into account by the caller of the original operation, by presuming they occur at least once during its execution. For calls that are not excluded, the possible Global effects of the call are those permitted by the Global aspect of the associated entity, or by its Global'Class aspect if a dispatching call. 37/5

If a Global aspect other than Unspecified or **in out all** applies to an access-to-subprogram type, then the prefix of an Access attribute_reference producing a value of such a type shall denote a subprogram whose Global aspect is not Unspecified and is *covered* by that of the result type, where a global aspect *G1* is *covered* by a global aspect *G2* if the set of variables that *G1* identifies as readable or updatable is a subset of the corresponding set for *G2*. Similarly on a conversion to such a type, the operand shall be of a named access-to-subprogram type whose Global aspect is covered by that of the target type. 38/5

If an implementation-defined global_mode applies to a given set of variables, an implementation-defined rule determines what sort of references to them are permitted. 39/5

For a subprogram that is a dispatching operation of a tagged type *T*, each mode of its Global aspect shall identify a subset of the variables identified by the corresponding mode, or by the **in out** mode, of the Global'Class aspect of a corresponding dispatching subprogram of any ancestor of *T*, unless the aspect of that ancestor is Unspecified. 40/5

Implementation Permissions

41/5 An implementation can allow some references to a constant object which are not accounted for by the Global or Global'Class aspect when it is considered a variable in the above rules, if the implementation can determine that the object is in fact immutable.

42/5 Implementations may perform additional checks on calls to operations with an Unspecified Global aspect to ensure that they do not violate any limitations associated with the point of call.

43/5 Implementations may extend the syntax or semantics of the Global aspect in an implementation-defined manner; for example, supporting additional global_modes.

44/5 NOTE For an example of the use of these aspects, see the Vector container definition in A.18.2.

6.2 Formal Parameter Modes

1 A parameter_specification declares a formal parameter of mode **in**, **in out**, or **out**.

Static Semantics

2 A parameter is passed either *by copy* or *by reference*. When a parameter is passed by copy, the formal parameter denotes a separate object from the actual parameter, and any information transfer between the two occurs only before and after executing the subprogram. When a parameter is passed by reference, the formal parameter denotes (a view of) the object denoted by the actual parameter; reads and updates of the formal parameter directly reference the actual parameter object.

3/3 A type is a *by-copy type* if it is an elementary type, or if it is a descendant of a private type whose full type is a by-copy type. A parameter of a by-copy type is passed by copy, unless the formal parameter is explicitly aliased.

4 A type is a *by-reference type* if it is a descendant of one of the following:

5 • a tagged type;

6 • a task or protected type;

7/3 • an explicitly limited record type;

8 • a composite type with a subcomponent of a by-reference type;

9 • a private type whose full type is a by-reference type.

10/5 A parameter of a by-reference type is passed by reference, as is an explicitly aliased parameter of any type. Each value of a by-reference type has an associated object. For a value conversion, the associated object is the anonymous result object if such an object is created (see 4.6); otherwise it is the associated object of the operand. In other cases, the object associated with the evaluated operative constituent of the name or expression (see 4.4) determines its associated object.

11/3 For other parameters, it is unspecified whether the parameter is passed by copy or by reference.

Bounded (Run-Time) Errors

12/3 If one name denotes a part of a formal parameter, and a second name denotes a part of a distinct formal parameter or an object that is not part of a formal parameter, then the two names are considered *distinct access paths*. If an object is of a type for which the parameter passing mechanism is not specified and is not an explicitly aliased parameter, then it is a bounded error to assign to the object via one access path, and then read the value of the object via a distinct access path, unless the first access path denotes a part of a formal parameter that no longer exists at the point of the second access (due to leaving the corresponding

callable construct). The possible consequences are that Program_Error is raised, or the newly assigned value is read, or some old value of the object is read.

NOTE 1 The mode of a formal parameter describes the direction of information transfer to or from the subprogram_body (see 6.1). 13/4

NOTE 2 A formal parameter of mode **in** is a constant view (see 3.3); it cannot be updated within the subprogram_body. 14/4

NOTE 3 A formal parameter of mode **out** can be uninitialized at the start of the subprogram_body (see 6.4.1). 15/5

6.3 Subprogram Bodies

A subprogram_body specifies the execution of a subprogram. 1

Syntax

subprogram_body ::= 2/3
[overriding_indicator]
subprogram_specification
[aspect_specification] **is**
declarative_part
begin
handled_sequence_of_statements
end [designator];

If a designator appears at the end of a subprogram_body, it shall repeat the defining_designator of the subprogram_specification. 3

Legality Rules

In contrast to other bodies, a subprogram_body is allowed to be defined without it being the completion of a previous declaration, in which case the body declares the subprogram. If the body is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body that completes a declaration shall conform fully to that of the declaration. 4/5

Static Semantics

A subprogram_body is considered a declaration. It can either complete a previous declaration, or itself be the initial declaration of the subprogram. 5

Dynamic Semantics

The elaboration of a nongeneric subprogram_body has no other effect than to establish that the subprogram can from then on be called without failing the Elaboration_Check. 6

The execution of a subprogram_body is invoked by a subprogram call. For this execution the declarative_part is elaborated, and the handled_sequence_of_statements is then executed. 7

Examples

8 *Example of procedure body:*

9
```
procedure Push(E : in Element_Type; S : in out Stack) is
begin
   if S.Index = S.Size then
      raise Stack_Overflow;
   else
      S.Index := S.Index + 1;
      S.Space(S.Index) := E;
   end if;
end Push;
```

10 *Example of a function body:*

11
```
function Dot_Product(Left, Right : Vector) return Real is
   Sum : Real := 0.0;
begin
   Check(Left'First = Right'First and Left'Last = Right'Last);
   for J in Left'Range loop
      Sum := Sum + Left(J)*Right(J);
   end loop;
   return Sum;
end Dot_Product;
```

6.3.1 Conformance Rules

1 When subprogram profiles are given in more than one place, they are required to conform in one of four ways: type conformance, mode conformance, subtype conformance, or full conformance.

Static Semantics

2/1 As explained in B.1, “Interfacing Aspects”, a *convention* can be specified for an entity. Unless this document states otherwise, the default convention of an entity is Ada. For a callable entity or access-to-subprogram type, the convention is called the *calling convention*. The following conventions are defined by the language:

3/3 - The default calling convention for any subprogram not listed below is *Ada*. The Convention aspect may be specified to override the default calling convention (see B.1).

4 - The *Intrinsic* calling convention represents subprograms that are “built in” to the compiler. The default calling convention is Intrinsic for the following:

5 - an enumeration literal;

6 - a "/=" operator declared implicitly due to the declaration of "=" (see 6.6);

7 - any other implicitly declared subprogram unless it is a dispatching operation of a tagged type;

8 - an inherited subprogram of a generic formal tagged type with unknown discriminants;

9 - an attribute that is a subprogram;

10/2 - a subprogram declared immediately within a protected_body;

10.1/4 - any prefixed view of a subprogram (see 4.1.3) without synchronization kind (see 9.5) By_Entry or By_Protected_Procedure.

11 The Access attribute is not allowed for Intrinsic subprograms.

12/4 - The default calling convention is *protected* for a protected subprogram, for a prefixed view of a subprogram with a synchronization kind of By_Protected_Procedure, and for an access-to-subprogram type with the reserved word **protected** in its definition.

- The default calling convention is *entry* for an entry and for a prefixed view of a subprogram with a synchronization kind of By_Entry. 13/4
- The calling convention for an anonymous access-to-subprogram parameter or anonymous access-to-subprogram result is *protected* if the reserved word **protected** appears in its definition; otherwise, it is the convention of the entity that has the parameter or result, unless that entity has convention *protected*, *entry*, or Intrinsic, in which case the convention is Ada. 13.1/5
- If not specified above as Intrinsic, the calling convention for any inherited or overriding dispatching operation of a tagged type is that of the corresponding subprogram of the parent type. The default calling convention for a new dispatching operation of a tagged type is the convention of the type. 13.2/1

Of these four conventions, only Ada and Intrinsic are allowed as a *convention*_identifier in the specification of a Convention aspect. 14/3

Two profiles are *type conformant* if they have the same number of parameters, and both have a result if either does, and corresponding parameter and result types are the same, or, for access parameters or access results, corresponding designated types are the same, or corresponding designated profiles are type conformant. 15/2

Two profiles are *mode conformant* if: 16/3

- they are type conformant; and 16.1/3
- corresponding parameters have identical modes and both or neither are explicitly aliased parameters; and 16.2/3
- for corresponding access parameters and any access result type, the designated subtypes statically match and either both or neither are access-to-constant, or the designated profiles are subtype conformant. 16.3/3

Two profiles are *subtype conformant* if they are mode conformant, corresponding subtypes of the profile statically match, and the associated calling conventions are the same. The profile of a generic formal subprogram is not subtype conformant with any other profile. 17/3

Two profiles are *fully conformant* if they are subtype conformant, if they have access-to-subprogram results whose designated profiles are fully conformant, and for corresponding parameters: 18/3

- they have the same names; and 18.1/3
- both or neither have null_exclusions; and 18.2/3
- neither have default_expressions, or they both have default_expressions that are fully conformant with one another; and 18.3/3
- for access-to-subprogram parameters, the designated profiles are fully conformant. 18.4/3

Two expressions are *fully conformant* if, after replacing each use of an operator with the equivalent function_call: 19

- each constituent construct of one corresponds to an instance of the same syntactic category in the other, except that an expanded name may correspond to a direct_name (or character_literal) or to a different expanded name in the other; and 20
- corresponding defining_identifiers occurring within the two expressions are the same; and 20.1/4
- each direct_name, character_literal, and selector_name that is not part of the prefix of an expanded name in one denotes the same declaration as the corresponding direct_name, character_literal, or selector_name in the other, they both denote type-conformant inherited 21/6

subprograms which are ultimately inherited from the same noninherited declaration, or they denote corresponding declarations occurring within the two expressions; and

21.1/3 • each attribute_designator in one is the same as the corresponding attribute_designator in the other; and

22/5 • each primary that is a literal in one is a user-defined literal if and only if the corresponding literal in the other is also a user-defined literal. Furthermore, if neither are user-defined literals then they shall have the same values, but they may have differing textual representations; if both are user-defined literals then they shall have the same textual representation.

23 Two known_discriminant_parts are *fully conformant* if they have the same number of discriminants, and discriminants in the same positions have the same names, statically matching subtypes, and default_expressions that are fully conformant with one another.

24 Two discrete_subtype_definitions are *fully conformant* if they are both subtype_indications or are both ranges, the subtype_marks (if any) denote the same subtype, and the corresponding simple_expressions of the ranges (if any) fully conform.

24.1/2 The *prefixed view profile* of a subprogram is the profile obtained by omitting the first parameter of that subprogram. There is no prefixed view profile for a parameterless subprogram. For the purposes of defining subtype and mode conformance, the convention of a prefixed view profile is considered to match that of either an entry or a protected operation.

Implementation Permissions

25 An implementation may declare an operator declared in a language-defined library unit to be intrinsic.

26/5 NOTE Any conformance requirements between aspect_specifications that are part of a profile or known_discriminant_part are defined by the semantics of each particular aspect. In particular, there is no general requirement for aspect_specifications to match in conforming profiles or discriminant parts.

6.3.2 Inline Expansion of Subprograms

1 Subprograms may be expanded in line at the call site.

Paragraphs 2 through 4 were moved to Annex J, "Obsolescent Features".

Static Semantics

5/3 For a callable entity or a generic subprogram, the following language-defined representation aspect may be specified:

5.1/3 Inline — The type of aspect Inline is Boolean. When aspect Inline is True for a callable entity, inline expansion is desired for all calls to that entity. When aspect Inline is True for a generic subprogram, inline expansion is desired for all calls to all instances of that generic subprogram.

5.2/3 If directly specified, the aspect_definition shall be a static expression. This aspect is never inherited; if not directly specified, the aspect is False.

Implementation Permissions

6/3 For each call, an implementation is free to follow or to ignore the recommendation determined by the Inline aspect.

6.4 Subprogram Calls

A *subprogram call* is either a procedure_call_statement or a function_call; it invokes the execution of the subprogram_body. The call specifies the association of the actual parameters, if any, with formal parameters of the subprogram. 1

Syntax

procedure_call_statement ::= 2
  *procedure*_name;
 | *procedure*_prefix actual_parameter_part;

function_call ::= 3
  *function*_name
 | *function*_prefix actual_parameter_part

actual_parameter_part ::= 4
  (parameter_association {, parameter_association})

parameter_association ::= 5
  [*formal_parameter*_selector_name =>] explicit_actual_parameter

explicit_actual_parameter ::= expression | *variable*_name 6

A parameter_association is *named* or *positional* according to whether or not the *formal_parameter*_selector_name is specified. For the parameter_associations of a single actual_parameter_part or iterator_actual_parameter_part, any positional associations shall precede any named associations. Named associations are not allowed if the prefix in a subprogram call is an attribute_reference. 7/5

Name Resolution Rules

The name or prefix given in a procedure_call_statement shall resolve to denote a callable entity that is a procedure, or an entry renamed as (viewed as) a procedure. The name or prefix given in a function_call shall resolve to denote a callable entity that is a function. The name or prefix shall not resolve to denote an abstract subprogram unless it is also a dispatching subprogram. When there is an actual_parameter_part, the prefix can be an implicit_dereference of an access-to-subprogram value. 8/2

A subprogram call shall contain at most one association for each formal parameter. Each formal parameter without an association shall have a default_expression (in the profile of the view denoted by the name or prefix). This rule is an overloading rule (see 8.6). 9

Static Semantics

If the name or prefix of a subprogram call denotes a prefixed view (see 4.1.3), the subprogram call is equivalent to a call on the underlying subprogram, with the first actual parameter being provided by the prefix of the prefixed view (or the Access attribute of this prefix if the first formal parameter is an access parameter), and the remaining actual parameters given by the actual_parameter_part, if any. In the case where the prefix is class-wide and controlling, if there are multiple controlling parameters, a check is made that they all have the same tag, as defined in 3.9.2. A call on a renaming of a prefixed view results in a dispatching call if the prefix of the prefixed view is class-wide and controlling. 9.1/6

Dynamic Semantics

10/2 For the execution of a subprogram call, the name or prefix of the call is evaluated, and each parameter_association is evaluated (see 6.4.1). If a default_expression is used, an implicit parameter_association is assumed for this rule. These evaluations are done in an arbitrary order. The subprogram_body is then executed, or a call on an entry or protected subprogram is performed (see 3.9.2). Finally, if the subprogram completes normally, then after it is left, any necessary assigning back of formal to actual parameters occurs (see 6.4.1).

11/2 The exception Program_Error is raised at the point of a function_call if the function completes normally without executing a return statement.

12/2 A function_call denotes a constant, as defined in 6.5; the nominal subtype of the constant is given by the nominal subtype of the function result.

Examples

13 *Examples of procedure calls:*

14
```
Traverse_Tree;                                                    -- see 6.1
Print_Header(128, Title, True);                                   -- see 6.1
```

15
```
Switch(From => X, To => Next);                                    -- see 6.1
Print_Header(128, Header => Title, Center => True);               -- see 6.1
Print_Header(Header => Title, Center => True, Pages => 128);      -- see 6.1
```

16 *Examples of function calls:*

17
```
Dot_Product(U, V)    -- see 6.1 and 6.3
Clock                -- see 9.6
F.all                -- presuming F is of an access-to-subprogram type — see 3.10
```

18 *Examples of procedures with default expressions:*

19
```
procedure Activate(Process : in Process_Name;
                   After   : in Process_Name := No_Process;
                   Wait    : in Duration := 0.0;
                   Prior   : in Boolean := False);
```

20/3
```
procedure Pair(Left, Right : in Person_Name := new Person(M));    -- see 3.10.1
```

21 *Examples of their calls:*

22
```
Activate(X);
Activate(X, After => Y);
Activate(X, Wait => 60.0, Prior => True);
Activate(X, Y, 10.0, False);
```

23/3
```
Pair;
Pair(Left => new Person(F), Right => new Person(M));
```

24 NOTE If a default_expression is used for two or more parameters in a multiple parameter_specification, the default_expression is evaluated once for each omitted parameter. Hence in the above examples, the two calls of Pair are equivalent.

Examples

25 *Examples of overloaded subprograms:*

26
```
procedure Put(X : in Integer);
procedure Put(X : in String);
```

27
```
procedure Set(Tint   : in Color);
procedure Set(Signal : in Light);
```

28 *Examples of their calls:*

29
```
Put(28);
Put("no possible ambiguity here");
```

```
Set(Tint   => Red);                                                    30
Set(Signal => Red);
Set(Color'(Red));
-- Set(Red) would be ambiguous since Red can                           31/5
-- denote a value either of type Color or of type Light
```

6.4.1 Parameter Associations

A parameter association defines the association between an actual parameter and a formal parameter. 1

Name Resolution Rules

The *formal_parameter*_selector_name of a named parameter_association shall resolve to denote a parameter_specification of the view being called; this is the formal parameter of the association. The formal parameter for a positional parameter_association is the parameter with the corresponding position in the formal part of the view being called. 2/3

The *actual parameter* is either the explicit_actual_parameter given in a parameter_association for a given formal parameter, or the corresponding default_expression if no parameter_association is given for the formal parameter. The expected type for an actual parameter is the type of the corresponding formal parameter. 3

If the mode is **in**, the actual is interpreted as an expression; otherwise, the actual is interpreted only as a name, if possible. 4

Legality Rules

If the mode is **in out** or **out**, the actual shall be a name that denotes a variable. 5

If the mode is **out**, the actual parameter is a view conversion, and the type of the formal parameter is a scalar type, then 5.1/5

- neither the target type nor the operand type has the Default_Value aspect specified; or 5.2/5
- both the target type and the operand type shall have the Default_Value aspect specified, and there shall exist a type (other than a root numeric type) that is an ancestor of both the target type and the operand type. 5.3/5

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 5.4/4

If the formal parameter is an explicitly aliased parameter, the type of the actual parameter shall be tagged or the actual parameter shall be an aliased view of an object. Further, if the formal parameter subtype *F* is untagged: 6/3

- the subtype *F* shall statically match the nominal subtype of the actual object; or 6.1/3
- the subtype *F* shall be unconstrained, discriminated in its full view, and unconstrained in any partial view. 6.2/3

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 6.3/4

In a function call, the accessibility level of the actual object for each explicitly aliased parameter shall not be statically deeper than the accessibility level of the master of the call (see 3.10.2). 6.4/3

Two names are *known to denote the same object* if: 6.5/3

- both names statically denote the same stand-alone object or parameter; or 6.6/3

6.7/3 • both names are selected_components, their prefixes are known to denote the same object, and their selector_names denote the same component; or

6.8/3 • both names are dereferences (implicit or explicit) and the dereferenced names are known to denote the same object; or

6.9/3 • both names are indexed_components, their prefixes are known to denote the same object, and each of the pairs of corresponding index values are either both static expressions with the same static value or both names that are known to denote the same object; or

6.10/3 • both names are slices, their prefixes are known to denote the same object, and the two slices have statically matching index constraints; or

6.11/3 • one of the two names statically denotes a renaming declaration whose renamed *object*_name is known to denote the same object as the other, the prefix of any dereference within the renamed *object*_name is not a variable, and any expression within the renamed *object*_name contains no references to variables nor calls on nonstatic functions.

6.12/3 Two names are *known to refer to the same object* if

6.13/3 • The two names are known to denote the same object; or

6.14/3 • One of the names is a selected_component, indexed_component, or slice and its prefix is known to refer to the same object as the other name; or

6.15/3 • One of the two names statically denotes a renaming declaration whose renamed *object*_name is known to refer to the same object as the other name.

6.16/3 If a call *C* has two or more parameters of mode **in out** or **out** that are of an elementary type, then the call is legal only if:

6.17/5 • For each name *N* denoting an object of an elementary type that is passed as a parameter of mode **in out** or **out** to the call *C*, there is no other name among the other parameters of mode **in out** or **out** to *C* that is known to denote the same object.

6.18/3 If a construct *C* has two or more direct constituents that are names or expressions whose evaluation may occur in an arbitrary order, at least one of which contains a function call with an **in out** or **out** parameter, then the construct is legal only if:

6.19/3 • For each name *N* that is passed as a parameter of mode **in out** or **out** to some inner function call *C2* (not including the construct *C* itself), there is no other name anywhere within a direct constituent of the construct *C* other than the one containing *C2*, that is known to refer to the same object.

6.20/3 For the purposes of checking this rule:

6.21/3 • For an array aggregate, an expression associated with a discrete_choice_list that has two or more discrete choices, or that has a nonstatic range, is considered as two or more separate occurrences of the expression;

6.22/3 • For a record aggregate:

6.23/3 — The expression of a record_component_association is considered to occur once for each associated component; and

6.24/3 — The default_expression for each record_component_association with <> for which the associated component has a default_expression is considered part of the aggregate;

6.25/3 • For a call, any default_expression evaluated as part of the call is considered part of the call.

Dynamic Semantics

For the evaluation of a parameter_association: 7

- The actual parameter is first evaluated. 8
- For an access parameter, the access_definition is elaborated, which creates the anonymous access type. 9
- For a parameter (of any mode) that is passed by reference (see 6.2), a view conversion of the actual parameter to the nominal subtype of the formal parameter is evaluated, and the formal parameter denotes that conversion. 10
- For an **in** or **in out** parameter that is passed by copy (see 6.2), the formal parameter object is created, and the value of the actual parameter is converted to the nominal subtype of the formal parameter and assigned to the formal. 11
- For an **out** parameter that is passed by copy, the formal parameter object is created, and: 12
  - For an access type, the formal parameter is initialized from the value of the actual, without checking whether the value satisfies any constraints, predicates, or null exclusions, but including any dynamic accessibility checks associated with a conversion to the type of the formal parameter. 13/5
  - For a scalar type that has the Default_Value aspect specified, the formal parameter is initialized from the value of the actual, without checking that the value satisfies any constraint or any predicate. 13.1/5
  - For a composite type with discriminants or that has implicit initial values for any subcomponents (see 3.3.1), the behavior is as for an **in out** parameter passed by copy, except that no predicate check is performed. 14/5
  - For any other type, the formal parameter is uninitialized. If composite, a view conversion of the actual parameter to the nominal subtype of the formal is evaluated (which can raise Constraint_Error), and the actual subtype of the formal is that of the view conversion. If elementary, the actual subtype of the formal is given by its nominal subtype. 15/5
  - Furthermore, if the type is a scalar type, and the actual parameter is a view conversion, then Program_Error is raised if either the target or the operand type has the Default_Value aspect specified, unless they both have the Default_Value aspect specified, and there is a type (other than a root numeric type) that is an ancestor of both the target type and the operand type. 15.1/5
- In a function call, for each explicitly aliased parameter, a check is made that the accessibility level of the master of the actual object is not deeper than that of the master of the call (see 3.10.2). 15.2/5

A formal parameter of mode **in out** or **out** with discriminants is constrained if either its nominal subtype or the actual parameter is constrained. 16

After normal completion and leaving of a subprogram, for each **in out** or **out** parameter that is passed by copy, the value of the formal parameter is converted to the subtype of the variable given as the actual parameter and assigned to it. These conversions and assignments occur in an arbitrary order. 17

Erroneous Execution

If the nominal subtype of a formal parameter with discriminants is constrained or indefinite, and the parameter is passed by reference, then the execution of the call is erroneous if the value of any discriminant of the actual is changed while the formal parameter exists (that is, before leaving the corresponding callable construct). 18/3

Implementation Permissions

19/5 If the actual parameter in a parameter_association with mode **out** is a view conversion between two access types that do not share a common ancestor type, the implementation may pass in the null value of the type of the formal parameter instead of the value of the actual parameter. It is implementation-defined under what circumstances the implementation passes in the null value.

6.5 Return Statements

A simple_return_statement or extended_return_statement (collectively called a *return statement*) is used to complete the execution of the innermost enclosing subprogram_body, entry_body, or accept_statement. 1/2

Syntax

simple_return_statement ::= **return** [expression]; 2/2

extended_return_object_declaration ::= 2.1/5
  defining_identifier : [**aliased**] [**constant**] return_subtype_indication [:= expression]
    [aspect_specification]

extended_return_statement ::= 2.2/3
  return extended_return_object_declaration [**do**
    handled_sequence_of_statements
  end return];

return_subtype_indication ::= subtype_indication | access_definition 2.3/2

Name Resolution Rules

The *result subtype* of a function is the subtype denoted by the subtype_mark, or defined by the access_definition, after the reserved word **return** in the profile of the function. The expected type for the expression, if any, of a simple_return_statement is the result type of the corresponding function. The expected type for the expression of an extended_return_object_declaration is that of the return_subtype_indication. 3/5

Legality Rules

A return statement shall be within a callable construct, and it *applies to* the innermost callable construct or extended_return_statement that contains it. A return statement shall not be within a body that is within the construct to which the return statement applies. 4/2

A function body shall contain at least one return statement that applies to the function body, unless the function contains code_statements. A simple_return_statement shall include an expression if and only if it applies to a function body. An extended_return_statement shall apply to a function body. An extended_return_object_declaration with the reserved word **constant** shall include an expression. 5/5

The *expression of an extended_return_statement* is the expression (if any) of the extended_return_object_declaration of the extended_return_statement. 5.1/5

For an extended_return_statement that applies to a function body: 5.2/2

- If the result subtype of the function is defined by a subtype_mark, the return_subtype_indication shall be a subtype_indication. The type of the subtype_indication shall be covered by the result type of the function. The subtype defined by the subtype_indication shall be statically compatible with the result subtype of the function; if the result type of the function is elementary, the two subtypes shall statically match. If the result subtype of the function is indefinite, then the subtype defined by the subtype_indication shall be a definite subtype, or there shall be an expression. 5.3/5
- If the result subtype of the function is defined by an access_definition, the return_subtype_indication shall be an access_definition. The subtype defined by the access_definition shall 5.4/5

statically match the result subtype of the function. The accessibility level of this anonymous access subtype is that of the result subtype.

5.5/5 • If the result subtype of the function is class-wide, the accessibility level of the type of the subtype defined by the return_subtype_indication shall not be statically deeper than that of the master that elaborated the function body.

5.6/5 For any return statement that applies to a function body:

5.7/5 • If the result subtype of the function is limited, then the expression of the return statement (if any) shall meet the restrictions described in 7.5.

5.8/5 • If the result subtype of the function is class-wide, the accessibility level of the type of the expression (if any) of the return statement shall not be statically deeper than that of the master that elaborated the function body.

5.9/5 • If the subtype determined by the expression of the simple_return_statement or by the return_subtype_indication has one or more access discriminants, the accessibility level of the anonymous access type of each access discriminant shall not be statically deeper than that of the master that elaborated the function body.

5.10/5 If the reserved word **aliased** is present in an extended_return_object_declaration, the type of the extended return object shall be immutably limited.

Static Semantics

5.11/5 Within an extended_return_statement, the *return object* is declared with the given defining_identifier, with the nominal subtype defined by the return_subtype_indication. An extended_return_statement with the reserved word **constant** is a full constant declaration that declares the return object to be a constant object.

Dynamic Semantics

5.12/5 For the execution of an extended_return_statement, the subtype_indication or access_definition is elaborated. This creates the nominal subtype of the return object. If there is an expression, it is evaluated and converted to the nominal subtype (which can raise Constraint_Error — see 4.6); the return object is created and the converted value is assigned to the return object. Otherwise, the return object is created and initialized by default as for a stand-alone object of its nominal subtype (see 3.3.1). If the nominal subtype is indefinite, the return object is constrained by its initial value. A check is made that the value of the return object belongs to the function result subtype. Constraint_Error is raised if this check fails.

6/2 For the execution of a simple_return_statement, the expression (if any) is first evaluated, converted to the result subtype, and then is assigned to the anonymous *return object*.

7/2 If the return object has any parts that are tasks, the activation of those tasks does not occur until after the function returns (see 9.2).

8/5 If the result type of a function is a specific tagged type, the tag of the return object is that of the result type. If the result type is class-wide, the tag of the return object is that of the value of the expression of the return statement, unless the return object is defined by an extended_return_object_declaration with a subtype_indication that is specific, in which case it is that of the type of the subtype_indication. A check is made that the master of the type identified by the tag of the result includes the elaboration of the master that elaborated the function body. If this check fails, Program_Error is raised.

8.1/5 For the execution of an extended_return_statement, the handled_sequence_of_statements is executed. Within this handled_sequence_of_statements, the execution of a simple_return_statement that applies to the extended_return_statement causes a transfer of control that completes the extended_return_-

statement. Upon completion of a return statement that applies to a callable construct by the normal completion of a simple_return_statement or by reaching the **end return** of an extended_return_statement, a transfer of control is performed which completes the execution of the callable construct, and returns to the caller.

If the result subtype of the function is defined by an access_definition designating a specific tagged type *T*, a check is made that the result value is null or the tag of the object designated by the result value identifies *T*. Constraint_Error is raised if this check fails. 8.2/5

Paragraphs 9 through 20 were deleted.

If any part of the specific type of the return object of a function (or coextension thereof) has one or more access discriminants whose value is not constrained by the result subtype of the function, a check is made that the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or the return_subtype_indication of the return statement, is not deeper than the level of the master of the call (see 3.10.2). If this check fails, Program_Error is raised. 21/3

A check is performed that the return value satisfies the predicates of the return subtype. If this check fails, the effect is as defined in 3.2.4. 22/5

In the case of a function, the function_call denotes a constant view of the return object. 23/2

Implementation Permissions

For a function call used to initialize a composite object with a constrained nominal subtype or used to initialize a return object that is built in place into such an object: 24/3

- If the result subtype of the function is constrained, and conversion of an object of this subtype to the subtype of the object being initialized would raise Constraint_Error, then Constraint_Error may be raised before calling the function. 24.1/3
- If the result subtype of the function is unconstrained, and a return statement is executed such that the return object is known to be constrained, and conversion of the return object to the subtype of the object being initialized would raise Constraint_Error, then Constraint_Error may be raised at the point of the call (after abandoning the execution of the function body). 24.2/3

Examples

Examples of return statements: 25

```
return;                                  -- in a procedure body, entry_body,
                                         --  accept_statement, or extended_return_statement   26/2

return Key_Value(Last_Index);            -- in a function body                                 27

return Node : Cell do                    -- in a function body, see 3.10.1 for Cell            28/2
   Node.Value := Result;
   Node.Succ := Next_Node;
end return;
```

6.5.1 Nonreturning Subprograms

1/5 Specifying aspect No_Return to have the value True indicates that a subprogram cannot return normally; it may, for example, propagate an exception or loop forever.

Paragraphs 2 and 3 were moved to Annex J, "Obsolescent Features".

Static Semantics

3.1/6 For a subprogram (including a generic formal subprogram), an access-to-subprogram type (including a formal access-to-subprogram type), or a generic subprogram, the following language-defined representation aspect may be specified:

3.2/3 No_Return The type of aspect No_Return is Boolean. When aspect No_Return is True for an entity, the entity is said to be *nonreturning*.

3.3/5 If directly specified, the aspect_definition shall be a static expression. When not directly specified, if the subprogram is a primitive subprogram inherited by a derived type, then the aspect is True if any corresponding subprogram of the parent or progenitor types is nonreturning. Otherwise, the aspect is False.

3.4/6 The No_Return aspect of a dereference of a value of an access-to-subprogram type is the same as that of the type.

3.5/6 If a generic subprogram is nonreturning, then so are its instances. If a subprogram declared within a generic unit is nonreturning, then so are the corresponding copies of that subprogram in instances.

Legality Rules

4/6 A subprogram defined by a null_procedure_declaration shall not be nonreturning. For an instance of a generic subprogram, any specification of aspect No_Return shall be confirming.

5/2 A return statement shall not apply to a nonreturning procedure or generic procedure.

5.1/6 An extended_return_statement shall not apply to a nonreturning function or generic function. Any return expression of a nonreturning function or generic function shall be a raise_expression, a call on a nonreturning function, or a parenthesized expression of one of these.

6/5 A subprogram shall be nonreturning if it overrides a dispatching nonreturning subprogram. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

7/5 If a renaming-as-body completes a nonreturning subprogram declaration, then the renamed subprogram shall be nonreturning.

7.1/6 If a No_Return aspect is True for a generic formal subprogram or a generic formal access-to-subprogram type, it shall be True for the matching actual in an instantiation.

7.2/6 If a No_Return aspect is True for an access-to-subprogram type *T*, then it shall be True for a subprogram *S* if *S*'Access has an expected type of *T*. Similarly, on a conversion to such a type, the No_Return aspect shall be true for the operand type.

Paragraph 8 was deleted.

Dynamic Semantics

9/2 If the body of a nonreturning procedure completes normally, Program_Error is raised at the point of the call.

Examples

Example of a specification of a No_Return aspect: 10/5

```
procedure Fail(Msg : String)   -- raises Fatal_Error exception
   with No_Return;
   -- Inform compiler and reader that procedure never returns normally
```
11/3

6.6 Overloading of Operators

An *operator* is a function whose designator is an operator_symbol. Operators, like other functions, may be overloaded. 1

Name Resolution Rules

Each use of a unary or binary operator is equivalent to a function_call with *function_*prefix being the corresponding operator_symbol, and with (respectively) one or two positional actual parameters being the operand(s) of the operator (in order). 2

Legality Rules

The subprogram_specification of a unary or binary operator shall have one or two parameters, respectively. The parameters shall be of mode **in**. A generic function instantiation whose designator is an operator_symbol is only allowed if the specification of the generic function has the corresponding number of parameters, and they are all of mode **in**. 3/3

Default_expressions are not allowed for the parameters of an operator (whether the operator is declared with an explicit subprogram_specification or by a generic_instantiation). 4

An explicit declaration of "/=" shall not have a result type of the predefined type Boolean. 5

Static Semantics

An explicit declaration of "=" whose result type is Boolean implicitly declares an operator "/=" that gives the complementary result. 6/3

NOTE The operators "+" and "–" are both unary and binary operators, and hence can be overloaded with both one- and two-parameter functions. 7/5

Examples

Examples of user-defined operators: 8

```
function "+" (Left, Right : Matrix) return Matrix;
function "+" (Left, Right : Vector) return Vector;

-- assuming that A, B, and C are of the type Vector
-- the following two statements are equivalent:

A := B + C;
A := "+"(B, C);
```
9

6.7 Null Procedures

1/2 A null_procedure_declaration provides a shorthand to declare a procedure with an empty body.

Syntax

2/3
```
null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification is null
     [aspect_specification];
```

Legality Rules

2.1/3 If a null_procedure_declaration is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a null_procedure_declaration that completes a declaration shall conform fully to that of the declaration.

Static Semantics

3/5 A null_procedure_declaration that is not a completion declares a *null procedure*. A completion is not allowed for a null_procedure_declaration; however, a null_procedure_declaration can complete a previous declaration.

Dynamic Semantics

4/5 The execution of a null procedure is invoked by a subprogram call. For the execution of a subprogram call on a null procedure, or on a procedure completed with a null_procedure_declaration, the execution of the subprogram_body has no effect.

5/3 The elaboration of a null_procedure_declaration has no other effect than to establish that the null procedure can be called without failing the Elaboration_Check.

Examples

6/5 *Example of the declaration of a null procedure:*

7/5
```
procedure Simplify(Expr : in out Expression) is null; -- see 3.9
-- By default, Simplify does nothing, but it can be overridden in extensions of Expression
```

6.8 Expression Functions

1/3 An expression_function_declaration provides a shorthand to declare a function whose body consists of a single return statement.

Syntax

2/4
```
expression_function_declaration ::=
  [overriding_indicator]
  function_specification is
     (expression)
     [aspect_specification];
 | [overriding_indicator]
  function_specification is
     aggregate
     [aspect_specification];
```

Name Resolution Rules

The expected type for the expression or aggregate of an expression_function_declaration is the result type (see 6.5) of the function. 3/4

Static Semantics

An expression_function_declaration that is not a completion declares an *expression function*. The *return expression of an expression function* is the expression or aggregate of the expression_function_declaration. A completion is not allowed for an expression_function_declaration; however, an expression_function_declaration can complete a previous declaration. 3.1/5

A *potentially static expression* is defined in the same way as a static expression except that 3.2/5

- a name denoting a formal parameter of an expression function is a potentially static expression; and 3.3/5
- each use of "static expression" in the definition of "static expression" is replaced with a corresponding use of "potentially static expression" in the definition of "potentially static expression". 3.4/5

The following language-defined representation aspect may be specified for an expression function: 3.5/5

Static — The type of aspect Static is Boolean. When aspect Static is True for an expression function, the function is a *static expression function*. If directly specified, the aspect_definition shall be a static expression. 3.6/5

The Static value for an inherited function is True if some corresponding primitive function of the parent or progenitor type is a static expression function; otherwise, if not directly specified, the aspect is False. 3.7/5

A static expression function is a static function; see 4.9. 3.8/5

Legality Rules

If an expression_function_declaration is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of an expression_function_declaration that completes a declaration shall conform fully to that of the declaration. 4/3

If the result subtype has one or more unconstrained access discriminants, the accessibility level of the anonymous access type of each access discriminant, as determined by the expression or aggregate of the expression_function_declaration, shall not be statically deeper than that of the master that elaborated the expression_function_declaration. 5/4

Aspect Static shall be specified to have the value True only if the associated expression_function_declaration: 5.1/5

- is not a completion; 5.2/5
- has an expression that is a potentially static expression; 5.3/5
- contains no calls to itself; 5.4/5
- each parameter (if any) is of mode **in** and is of a static subtype; 5.5/5
- has a result subtype that is a static subtype; 5.6/5
- has no applicable precondition or postcondition expression; and 5.7/5
- for result type *R*, if the function is a boundary entity for type *R* (see 7.3.2), no type invariant applies to type *R*; if *R* has a component type *C*, a similar rule applies to *C*. 5.8/5

Paragraph 6 was deleted.

Dynamic Semantics

7/5 The execution of an expression function is invoked by a subprogram call. For the execution of a subprogram call on an expression function, or on a function completed with a expression_function_declaration, the execution of the subprogram_body executes an implicit function body containing only a simple_return_statement whose expression is the return expression of the expression function.

8/3 The elaboration of an expression_function_declaration has no other effect than to establish that the expression function can be called without failing the Elaboration_Check.

Examples

9/5 *Example of an expression function:*

10/3
```
function Is_Origin (P : in Point) return Boolean is -- see 3.9
   (P.X = 0.0 and P.Y = 0.0);
```

7 Packages

Packages are program units that allow the specification of groups of logically related entities. Typically, a package contains the declaration of a type (often a private type or private extension) along with the declarations of primitive subprograms of the type, which can be called from outside the package, while their inner workings remain hidden from outside users. 1

7.1 Package Specifications and Declarations

A package is generally provided in two parts: a package_specification and a package_body. Every package has a package_specification, but not all packages have a package_body. 1

Syntax

package_declaration ::= package_specification; 2

```
package_specification ::=                                          3/3
   package defining_program_unit_name
      [aspect_specification] is
     {basic_declarative_item}
  [private
     {basic_declarative_item}]
   end [[parent_unit_name.]identifier]
```

If an identifier or parent_unit_name.identifier appears at the end of a package_specification, then this sequence of lexical elements shall repeat the defining_program_unit_name. 4

Legality Rules

A package_declaration or generic_package_declaration requires a completion (a body) if it contains any basic_declarative_item that requires a completion, but whose completion is not in its package_specification. 5/2

Static Semantics

The first list of basic_declarative_items of a package_specification of a package other than a generic formal package is called the *visible part* of the package. The optional list of basic_declarative_items after the reserved word **private** (of any package_specification) is called the *private part* of the package. If the reserved word **private** does not appear, the package has an implicit empty private part. Each list of basic_declarative_items of a package_specification forms a *declaration list* of the package. 6/2

An entity declared in the private part of a package is visible only within the declarative region of the package itself (including any child units — see 10.1.1). In contrast, expanded names denoting entities declared in the visible part can be used even outside the package; furthermore, direct visibility of such entities can be achieved by means of use_clauses (see 4.1.3 and 8.4). 7

Dynamic Semantics

The elaboration of a package_declaration consists of the elaboration of its basic_declarative_items in the given order. 8

NOTE 1 The visible part of a package contains all the information that another program unit is able to know about the package. 9

10 NOTE 2 If a declaration occurs immediately within the specification of a package, and the declaration has a corresponding completion that is a body, then that body has to occur immediately within the body of the package.

Examples

11 *Example of a package declaration:*

```
12    package Rational_Numbers is
13       type Rational is
            record
               Numerator   : Integer;
               Denominator : Positive;
            end record;
14       function "="(X,Y : Rational) return Boolean;
15       function "/"  (X,Y : Integer)  return Rational;   -- to construct a rational number
16       function "+"  (X,Y : Rational) return Rational;
         function "-"  (X,Y : Rational) return Rational;
         function "*"  (X,Y : Rational) return Rational;
         function "/"  (X,Y : Rational) return Rational;
    end Rational_Numbers;
```

17 There are also many examples of package declarations in the predefined language environment (see Annex A).

7.2 Package Bodies

1 In contrast to the entities declared in the visible part of a package, the entities declared in the package_body are visible only within the package_body itself. As a consequence, a package with a package_body can be used for the construction of a group of related subprograms in which the logical operations available to clients are clearly isolated from the internal entities.

Syntax

2/3 package_body ::=
package body defining_program_unit_name
[aspect_specification] **is**
declarative_part
[**begin**
handled_sequence_of_statements]
end [[parent_unit_name.]identifier];

3 If an identifier or parent_unit_name.identifier appears at the end of a package_body, then this sequence of lexical elements shall repeat the defining_program_unit_name.

Legality Rules

4/5 A package_body shall be the completion of a previous package_declaration or generic_package_declaration. A library package_declaration or library generic_package_declaration shall not have a body unless it requires a body; the Elaborate_Body aspect can be used to require a library_unit_declaration to have a body (see 10.2.1) if it would not otherwise require one.

Static Semantics

5 In any package_body without statements there is an implicit null_statement. For any package_declaration without an explicit completion, there is an implicit package_body containing a single null_statement. For a noninstance, nonlibrary package, this body occurs at the end of the declarative_part

of the innermost enclosing program unit or block_statement; if there are several such packages, the order of the implicit package_bodies is unspecified. (For an instance, the implicit package_body occurs at the place of the instantiation (see 12.3). For a library package, the place is partially determined by the elaboration dependences (see Clause 10).)

Dynamic Semantics

For the elaboration of a nongeneric package_body, its declarative_part is first elaborated, and its handled_sequence_of_statements is then executed. 6

NOTE 1 A variable declared in the body of a package is only visible within this body and, consequently, its value can only be changed within the package_body. In the absence of local tasks, the value of such a variable remains unchanged between calls issued from outside the package to subprograms declared in the visible part. The properties of such a variable are similar to those of a "static" variable of C. 7

NOTE 2 The elaboration of the body of a subprogram explicitly declared in the visible part of a package is caused by the elaboration of the body of the package. Hence a call of such a subprogram by an outside program unit raises the exception Program_Error if the call takes place before the elaboration of the package_body (see 3.11). 8

Examples

Example of a package body (see 7.1): 9

```
package body Rational_Numbers is                                                    10
   procedure Same_Denominator (X,Y : in out Rational) is                            11
   begin
      -- reduces X and Y to the same denominator:
      ...
   end Same_Denominator;
   function "="(X,Y : Rational) return Boolean is                                   12
      U : Rational := X;
      V : Rational := Y;
   begin
      Same_Denominator (U,V);
      return U.Numerator = V.Numerator;
   end "=";
   function "/" (X,Y : Integer) return Rational is                                  13
   begin
      if Y > 0 then
         return (Numerator => X,  Denominator => Y);
      else
         return (Numerator => -X, Denominator => -Y);
      end if;
   end "/";
   function "+" (X,Y : Rational) return Rational is ... end "+";                    14
   function "-" (X,Y : Rational) return Rational is ... end "-";
   function "*" (X,Y : Rational) return Rational is ... end "*";
   function "/" (X,Y : Rational) return Rational is ... end "/";
end Rational_Numbers;                                                               15
```

7.3 Private Types and Private Extensions

The declaration (in the visible part of a package) of a type as a private type or private extension serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself). See 3.9.1 for an overview of type extensions. 1

Syntax

2/3 private_type_declaration ::=
type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**
[aspect_specification];

3/3 private_extension_declaration ::=
type defining_identifier [discriminant_part] **is**
[**abstract**] [**limited** | **synchronized**] **new** *ancestor*_subtype_indication
[**and** interface_list] **with private**
[aspect_specification];

Legality Rules

4 A private_type_declaration or private_extension_declaration declares a *partial view* of the type; such a declaration is allowed only as a declarative_item of the visible part of a package, and it requires a completion, which shall be a full_type_declaration that occurs as a declarative_item of the private part of the package. The view of the type declared by the full_type_declaration is called the *full view*. A generic formal private type or a generic formal private extension is also a partial view.

5 A type shall be completely defined before it is frozen (see 3.11.1 and 13.14). Thus, neither the declaration of a variable of a partial view of a type, nor the creation by an allocator of an object of the partial view are allowed before the full declaration of the type. Similarly, before the full declaration, the name of the partial view cannot be used in a generic_instantiation or in a representation item.

6/2 A private type is limited if its declaration includes the reserved word **limited**; a private extension is limited if its ancestor type is a limited type that is not an interface type, or if the reserved word **limited** or **synchronized** appears in its definition. If the partial view is nonlimited, then the full view shall be nonlimited. If a tagged partial view is limited, then the full view shall be limited. On the other hand, if an untagged partial view is limited, the full view may be limited or nonlimited.

7 If the partial view is tagged, then the full view shall be tagged. On the other hand, if the partial view is untagged, then the full view may be tagged or untagged. In the case where the partial view is untagged and the full view is tagged, no derivatives of the partial view are allowed within the immediate scope of the partial view; derivatives of the full view are allowed.

7.1/2 If a full type has a partial view that is tagged, then:

7.2/2 • the partial view shall be a synchronized tagged type (see 3.9.4) if and only if the full type is a synchronized tagged type;

7.3/2 • the partial view shall be a descendant of an interface type (see 3.9.4) if and only if the full type is a descendant of the interface type.

8 The *ancestor subtype* of a private_extension_declaration is the subtype defined by the *ancestor*_subtype_indication; the ancestor type shall be a specific tagged type. The full view of a private extension shall be derived (directly or indirectly) from the ancestor type. In addition to the places where Legality Rules normally apply (see 12.3), the requirement that the ancestor be specific applies also in the private part of an instance of a generic unit.

8.1/2 If the reserved word **limited** appears in a private_extension_declaration, the ancestor type shall be a limited type. If the reserved word **synchronized** appears in a private_extension_declaration, the ancestor type shall be a limited interface.

If the declaration of a partial view includes a known_discriminant_part, then the full_type_declaration shall have a fully conforming (explicit) known_discriminant_part (see 6.3.1). The ancestor subtype may be unconstrained; the parent subtype of the full view is required to be constrained (see 3.7). 9/5

If a private extension inherits known discriminants from the ancestor subtype, then the full view shall also inherit its discriminants from the ancestor subtype, and the parent subtype of the full view shall be constrained if and only if the ancestor subtype is constrained. 10

If the full_type_declaration for a private extension includes a derived_type_definition, then the reserved word **limited** shall appear in the full_type_declaration if and only if it also appears in the private_extension_declaration. 10.1/3

If a partial view has unknown discriminants, then the full_type_declaration may define a definite or an indefinite subtype, with or without discriminants. 11

If a partial view has neither known nor unknown discriminants, then the full_type_declaration shall define a definite subtype. 12

If the ancestor subtype of a private extension has constrained discriminants, then the parent subtype of the full view shall impose a statically matching constraint on those discriminants. 13

Each predicate specification that applies to some ancestor of a private extension shall also apply to some ancestor of the full view. 13.1/6

Static Semantics

A private_type_declaration declares a private type and its first subtype. Similarly, a private_extension_-declaration declares a private extension and its first subtype. 14

A declaration of a partial view and the corresponding full_type_declaration define two views of a single type. The declaration of a partial view together with the visible part define the operations that are available to outside program units; the declaration of the full view together with the private part define other operations whose direct use is possible only within the declarative region of the package itself. Moreover, within the scope of the declaration of the full view, the characteristics (see 3.4) of the type are determined by the full view; in particular, within its scope, the full view determines the classes that include the type, which components, entries, and protected subprograms are visible, what attributes and other predefined operations are allowed, and whether the first subtype is static. See 7.3.1. 15/3

For a private extension, the characteristics (including components, but excluding discriminants if there is a new discriminant_part specified), predefined operators, and inherited user-defined primitive subprograms are determined by its ancestor type and its progenitor types (if any), in the same way that those of a record extension are determined by those of its parent type and its progenitor types (see 3.4 and 7.3.1). 16/3

Dynamic Semantics

The elaboration of a private_type_declaration creates a partial view of a type. The elaboration of a private_extension_declaration elaborates the *ancestor*_subtype_indication, and creates a partial view of a type. 17

NOTE 1 The partial view of a type as declared by a private_type_declaration is defined to be a composite view (in 3.2). The full view of the type can be elementary or composite. A private extension is also composite, as is its full view. 18/5

NOTE 2 Declaring a private type with an unknown_discriminant_part is a way of preventing clients from creating uninitialized objects of the type; they are then forced to initialize each object by calling some operation declared in the visible part of the package. 19/2

20/5 NOTE 3 The ancestor type specified in a private_extension_declaration and the parent type specified in the corresponding declaration of a record extension given in the private part can be different. If the ancestor type is not an interface type, the parent type of the full view can be any descendant of the ancestor type. In this case, for a primitive subprogram that is inherited from the ancestor type and not overridden, the formal parameter names and default expressions (if any) come from the corresponding primitive subprogram of the specified ancestor type, while the body comes from the corresponding primitive subprogram of the parent type of the full view. See 3.9.2.

21/5 NOTE 4 If the ancestor type specified in a private_extension_declaration is an interface type, the parent type can be any type so long as the full view is a descendant of the ancestor type. The progenitor types specified in a private_extension_declaration and the progenitor types specified in the corresponding declaration of a record extension given in the private part are not necessarily the same — it is only necessary that the private extension and the record extension be descended from the same set of interfaces.

Examples

22 *Examples of private type declarations:*

23
```
type Key is private;
type File_Name is limited private;
```

24 *Example of a private extension declaration:*

25
```
type List is new Ada.Finalization.Controlled with private;
```

7.3.1 Private Operations

1 For a type declared in the visible part of a package or generic package, certain operations on the type do not become visible until later in the package — either in the private part or the body. Such *private operations* are available only inside the declarative region of the package or generic package.

Static Semantics

2 The predefined operators that exist for a given type are determined by the classes to which the type belongs. For example, an integer type has a predefined "+" operator. In most cases, the predefined operators of a type are declared immediately after the definition of the type; the exceptions are explained below. Inherited subprograms are also implicitly declared immediately after the definition of the type, except as stated below.

3/3 For a composite type, the characteristics (see 7.3) of the type are determined in part by the characteristics of its component types. At the place where the composite type is declared, the only characteristics of component types used are those characteristics visible at that place. If later immediately within the declarative region in which the composite type is declared additional characteristics become visible for a component type, then any corresponding characteristics become visible for the composite type. Any additional predefined operators are implicitly declared at that place. If there is no such place, then additional predefined operators are not declared at all, but they still exist.

4/1 The corresponding rule applies to a type defined by a derived_type_definition, if there is a place immediately within the declarative region in which the type is declared where additional characteristics of its parent type become visible.

5/1 For example, an array type whose component type is limited private becomes nonlimited if the full view of the component type is nonlimited and visible at some later place immediately within the declarative region in which the array type is declared. In such a case, the predefined "=" operator is implicitly declared at that place, and assignment is allowed after that place.

5.1/5 The characteristics and constraints of the designated subtype of an access type follow a somewhat different rule. The view of the designated subtype of (a view of) an access type at a given place is

determined by the view of the designated subtype that is visible at that place, rather than the view at the place where the access type is declared.

A type is a *descendant* of the full view of some ancestor of its parent type only if the current view it has of its parent is a descendant of the full view of that ancestor. More generally, at any given place, a type is descended from the same view of an ancestor as that from which the current view of its parent is descended. This view determines what characteristics are inherited from the ancestor, and, for example, whether the type is considered to be a descendant of a record type, or a descendant only through record extensions of a more distant ancestor. 5.2/5

Furthermore, it is possible for there to be places where a derived type is known to be derived indirectly from an ancestor type, but is not a descendant of even a partial view of the ancestor type, because the parent of this derived type is not visibly a descendant of the ancestor. In this case, the derived type inherits no characteristics from that ancestor, but nevertheless is within the derivation class of the ancestor for the purposes of type conversion, the "covers" relationship, and matching against a formal derived type. In this case the derived type is effectively a *descendant* of an incomplete view of the ancestor. 5.3/5

Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, immediately within the declarative region in which the type_declaration occurs, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all, but it still exists. For a tagged type, it is possible to dispatch to an inherited subprogram that is not declared at all. 6/3

For a private_extension_declaration, each inherited subprogram is declared immediately after the private_extension_declaration if the corresponding declaration from the ancestor is visible at that place. Otherwise, the inherited subprogram is not declared for the private extension, though it can be for the full type. 7/5

The Class attribute is defined for tagged subtypes in 3.9. In addition, for every subtype S of an untagged private type whose full view is tagged, the following attribute is defined: 8

S'Class Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. After the full view, the Class attribute of the full view can be used. 9

NOTE 1 Because a partial view and a full view are two different views of one and the same type, outside of the defining package the characteristics of the type are those defined by the visible part. Within these outside program units the type is just a private type or private extension, and any language rule that applies only to another class of types does not apply. The fact that the full declaration can implement a private type with a type of a particular class (for example, as an array type) is relevant only within the declarative region of the package itself including any child units. 10

The consequences of this actual implementation are, however, valid everywhere. For example: any default initialization of components takes place; the attribute Size provides the size of the full view; finalization is still done for controlled components of the full view; task dependence rules still apply to components that are task objects. 11

NOTE 2 Partial views provide initialization, membership tests, selected components for the selection of discriminants and inherited components, qualification, and explicit conversion. Nonlimited partial views also allow use of assignment_statements. 12/2

NOTE 3 For a subtype S of a partial view, S'Size is defined (see 13.3). For an object A of a partial view, the attributes A'Size and A'Address are defined (see 13.3). The Position, First_Bit, and Last_Bit attributes are also defined for discriminants and inherited components. 13

Examples

14 *Example of a type with private operations:*

15
```
package Key_Manager is
   type Key is private;
   Null_Key : constant Key; -- a deferred constant declaration (see 7.4)
   procedure Get_Key(K : out Key);
   function "<" (X, Y : Key) return Boolean;
private
   type Key is new Natural;
   Null_Key : constant Key := Key'First;
end Key_Manager;
```

16
```
package body Key_Manager is
   Last_Key : Key := Null_Key;
   procedure Get_Key(K : out Key) is
   begin
      Last_Key := Last_Key + 1;
      K := Last_Key;
   end Get_Key;
```

17
```
   function "<" (X, Y : Key) return Boolean is
   begin
      return Natural(X) < Natural(Y);
   end "<";
end Key_Manager;
```

18/5 Outside of the package Key_Manager, the operations available for objects of type Key include assignment, the comparison for equality or inequality, the procedure Get_Key and the operator "<"; they do not include other relational operators such as ">=", or arithmetic operators.

19/5 The explicitly declared operator "<" hides the predefined operator "<" implicitly declared by the full_type_declaration. Within the body of the function, an explicit conversion of X and Y to the subtype Natural is necessary to invoke the "<" operator of the parent type. Alternatively, the result of the function can be written as **not** (X >= Y), since the operator ">=" is not redefined.

20/5 The value of the variable Last_Key, declared in the package body, remains unchanged between calls of the procedure Get_Key. (See also the NOTEs of 7.2.)

7.3.2 Type Invariants

0.1/6 A type invariant for a given private type is an assertion that is expected to be true about every object of the type, except when such an object is in the middle of an operation that has visibility on the representation of its full type. The type invariant is enforced much like a postcondition, including when such an object completes default initialization, or upon return from an externally callable subprogram that can update the object and has visibility on the full type.

1/5 For a private type, private extension, or interface, the following language-defined assertion aspects may be specified with an aspect_specification (see 13.1.1):

2/3 Type_Invariant

This aspect shall be specified by an expression, called an *invariant expression*. Type_Invariant may be specified on a private_type_declaration, on a private_extension_declaration, or on a full_type_declaration that declares the completion of a private type or private extension.

3/5 Type_Invariant'Class

This aspect shall be specified by an expression, called an *invariant expression*. Type_Invariant'Class may be specified on a private_type_declaration, a private_extension_declaration, or a full_type_declaration for an interface type.

Type_Invariant'Class determines a *class-wide type invariant* for a tagged type. The Type_Invariant'Class aspect is not inherited, but its effects are additive, as defined below.

Name Resolution Rules

The expected type for an invariant expression is any boolean type. 4/3

Within an invariant expression, a name that denotes the associated type declaration is interpreted as denoting the current instance of the type. Within an invariant expression for the Type_Invariant aspect of a type *T*, the type of this current instance is *T*. Within an invariant expression for the Type_Invariant'Class aspect of a type *T*, the type of this current instance is interpreted as though it had a (notional) nonabstract type *NT* that is a visible formal derived type whose ancestor type is *T*. The effect of this interpretation is that the only operations that can be applied to this current instance are those defined for such a formal derived type. 5/6

Legality Rules

The Type_Invariant'Class aspect shall not be specified for an untagged type. The Type_Invariant aspect shall not be specified for an abstract type. 6/3

If a type extension occurs immediately within the visible part of a package specification, at a point where a private operation of some ancestor is visible and inherited, and a Type_Invariant'Class expression applies to that ancestor, then the inherited operation shall be abstract or shall be overridden. 6.1/5

Static Semantics

If the Type_Invariant aspect is specified for a type *T*, then the invariant expression applies to *T*. 7/3

If the Type_Invariant'Class aspect is specified for a tagged type *T*, then a *corresponding expression* also applies to each nonabstract descendant *T1* of *T* (including *T* itself if it is nonabstract). The corresponding expression is constructed from the associated expression as follows: 8/5

- References to nondiscriminant components of *T* (or to *T* itself) are replaced with references to the corresponding components of *T1* (or to *T1* as a whole). 8.1/5
- References to discriminants of *T* are replaced with references to the corresponding discriminant of *T1*, or to the specified value for the discriminant, if the discriminant is specified by the derived_type_definition for some type that is an ancestor of *T1* and a descendant of *T* (see 3.7). 8.2/5

For a nonabstract type *T*, a callable entity is said to be a *boundary entity* for *T* if it is declared within the immediate scope of *T* (or by an instance of a generic unit, and the generic is declared within the immediate scope of type *T*), or is the Read or Input stream-oriented attribute of type *T*, and either: 8.3/5

- *T* is a private type or a private extension and the callable entity is visible outside the immediate scope of type T or overrides an inherited operation that is visible outside the immediate scope of *T*; or 8.4/5
- *T* is a record extension, and the callable entity is a primitive operation visible outside the immediate scope of type *T* or overrides an inherited operation that is visible outside the immediate scope of *T*. 8.5/5

Dynamic Semantics

If one or more invariant expressions apply to a nonabstract type *T*, then an invariant check is performed at the following places, on the specified object(s): 9/4

- After successful initialization of an object of type *T* by default (see 3.3.1), the check is performed on the new object unless the partial view of *T* has unknown discriminants; 10/4

10.1/5 • After successful explicit initialization of the completion of a deferred constant whose nominal type has a part of type *T*, if the completion is inside the immediate scope of the full view of *T*, and the deferred constant is visible outside the immediate scope of *T*, the check is performed on the part(s) of type *T*;

11/3 • After successful conversion to type *T*, the check is performed on the result of the conversion;

12/3 • For a view conversion, outside the immediate scope of *T*, that converts from a descendant of *T* (including *T* itself) to an ancestor of type *T* (other than *T* itself), a check is performed on the part of the object that is of type *T*:

13/3 • after assigning to the view conversion; and

14/3 • after successful return from a call that passes the view conversion as an **in out** or **out** parameter.

15/5 • Upon successful return from a call on any callable entity which is a boundary entity for *T*, an invariant check is performed on each object which is subject to an invariant check for *T*. In the case of a call to a protected operation, the check is performed before the end of the protected action. In the case of a call to a task entry, the check is performed before the end of the rendezvous. The following objects of a callable entity are subject to an invariant check for *T*:

Paragraph 16 was merged above.

17/5 • a result with a nominal type that has a part of type *T*;

18/5 • an **out** or **in out** parameter whose nominal type has a part of type *T*;

18.1/5 • an access-to-object parameter or result whose designated nominal type has a part of type *T*; or

19/5 • for a procedure or entry, an **in** parameter whose nominal type has a part of type *T*.

20/5 If the nominal type of a formal parameter (or the designated nominal type of an access-to-object parameter or result) is incomplete at the point of the declaration of the callable entity, and if the completion of that incomplete type does not occur in the same declaration list as the incomplete declaration, then for purposes of the preceding rules the nominal type is considered to have no parts of type *T*.

20.1/4 • For a view conversion to a class-wide type occurring within the immediate scope of *T*, from a specific type that is a descendant of *T* (including *T* itself), a check is performed on the part of the object that is of type *T*.

21/4 If performing checks is required by the Type_Invariant or Type_Invariant'Class assertion policies (see 11.4.2) in effect at the point of the corresponding aspect specification applicable to a given type, then the respective invariant expression is considered *enabled*.

22/3 The invariant check consists of the evaluation of each enabled invariant expression that applies to *T*, on each of the objects specified above. If any of these evaluate to False, Assertions.Assertion_Error is raised at the point of the object initialization, conversion, or call. If a given call requires more than one evaluation of an invariant expression, either for multiple objects of a single type or for multiple types with invariants, the evaluations are performed in an arbitrary order, and if one of them evaluates to False, it is not specified whether the others are evaluated. Any invariant check is performed prior to copying back any by-copy **in out** or **out** parameters. Invariant checks, any postcondition check, and any constraint or predicate checks associated with **in out** or **out** parameters are performed in an arbitrary order.

22.1/4 For an invariant check on a value of type *T1* based on a class-wide invariant expression inherited from an ancestor type *T*, any operations within the invariant expression that were resolved as primitive operations of the (notional) formal derived type *NT* are bound to the corresponding operations of type *T1* in the evaluation of the invariant expression for the check on *T1*.

The invariant checks performed on a call are determined by the subprogram or entry actually invoked, whether directly, as part of a dispatching call, or as part of a call through an access-to-subprogram value. 23/3

NOTE For a call of a primitive subprogram of type *NT* that is inherited from type *T*, the specified checks of the specific invariants of both the types *NT* and *T* are performed. For a call of a primitive subprogram of type *NT* that is overridden for type *NT*, the specified checks of the specific invariants of only type *NT* are performed. 24/3

Examples

Example of a work scheduler where only urgent work can be scheduled for weekends: 25/5

```
package Work_Orders is                                                                  26/5
   -- See 3.5.1 for type declarations of Level, Day, and Weekday                       27/5
   type Work_Order is private with                                                      28/5
     Type_Invariant => Day_Scheduled (Work_Order) in Weekday
                       or else Priority (Work_Order) = Urgent;
   function Schedule_Work (Urgency  : in Level;                                         29/5
                           To_Occur : in Day) return Work_Order
     with Pre => Urgency = Urgent or else To_Occur in Weekday;
   function Day_Scheduled (Order : in Work_Order) return Day;                           30/5
   function Priority (Order : in Work_Order) return Level;                              31/5
   procedure Change_Priority (Order        : in out Work_Order;                         32/5
                              New_Priority : in     Level;
                              Changed      : out    Boolean)
      with Post => Changed = (Day_Scheduled(Order) in Weekday
                              or else Priority(Order) = Urgent);
private                                                                                 33/5
   type Work_Order is record                                                            34/5
      Scheduled : Day;
      Urgency   : Level;
   end record;
end Work_Orders;                                                                        35/5
package body Work_Orders is                                                             36/5
   function Schedule_Work (Urgency  : in Level;                                         37/5
                           To_Occur : in Day) return Work_Order is
     (Scheduled => To_Occur, Urgency => Urgency);
   function Day_Scheduled (Order : in Work_Order) return Day is                         38/5
     (Order.Scheduled);
   function Priority (Order : in Work_Order) return Level is                            39/5
     (Order.Urgency);
   procedure Change_Priority (Order        : in out Work_Order;                         40/5
                              New_Priority : in     Level;
                              Changed      : out    Boolean) is
   begin
      -- Ensure type invariant is not violated
      if Order.Urgency = Urgent or else (Order.Scheduled in Weekday) then
         Changed := True;
         Order.Urgency := New_Priority;
      else
         Changed := False;
      end if;
   end Change_Priority;
end Work_Orders;                                                                        41/5
```

7.3.3 Default Initial Conditions

1/5 For a private type or private extension (including a generic formal type), the following language-defined assertion aspect may be specified with an aspect_specification (see 13.1.1):

2/5 Default_Initial_Condition

This aspect shall be specified by an expression, called a *default initial condition expression*. Default_Initial_Condition may be specified on a private_type_declaration, a private_extension_declaration, a formal_private_type_definition, or a formal_derived_type_definition. The Default_Initial_Condition aspect is not inherited, but its effects are additive, as defined below.

Name Resolution Rules

3/5 The expected type for a default initial condition expression is any boolean type.

4/5 Within a default initial condition expression associated with a declaration for a type T, a name that denotes the declaration is interpreted as a current instance of a notional (nonabstract) formal derived type NT with ancestor T, that has directly visible primitive operations.

Legality Rules

5/5 The Default_Initial_Condition aspect shall not be specified for a type whose partial view has unknown discriminants, whether explicitly declared or inherited.

Static Semantics

6/5 If the Default_Initial_Condition aspect is specified for a type T, then the default initial condition expression applies to T and to all descendants of T.

Dynamic Semantics

7/5 If one or more default initial condition expressions apply to a (nonabstract) type T, then a default initial condition check is performed after successful initialization of an object of type T by default (see 3.3.1). In the case of a controlled type, the check is performed after the call to the type's Initialize procedure (see 7.6).

8/5 If performing checks is required by the Default_Initial_Condition assertion policy (see 11.4.2) in effect at the point of the corresponding aspect_specification applicable to a given type, then the respective default initial condition expression is considered enabled.

9/5 The default initial condition check consists of the evaluation of each enabled default initial condition expression that applies to T. Any operations within such an expression that were resolved as primitive operations of the (notional) formal derived type NT, are in the evaluation of the expression resolved as for a formal derived type in an instance with T as the actual type for NT (see 12.5.1). These evaluations, if there are more than one, are performed in an arbitrary order. If any of these evaluate to False, Assertions.Assertion_Error is raised at the point of the object initialization.

10/5 For a generic formal type T, default initial condition checks performed are as determined by the actual type, along with any default initial condition of the formal type itself.

Implementation Permissions

11/5 Implementations may extend the syntax or semantics of the Default_Initial_Condition aspect in an implementation-defined manner.

12/5 NOTE For an example of the use of this aspect, see the Vector container definition in A.18.2.

7.3.4 Stable Properties of a Type

Certain characteristics of an object of a given type are unchanged by most of the primitive operations of the type. Such characteristics are called *stable properties* of the type. 1/5

Static Semantics

A *property function* of type *T* is a function with a single parameter of type *T* or of a class-wide type that covers *T*. 2/5

A *type property aspect definition* is a list of names written in the syntax of a positional_array_aggregate. A *subprogram property aspect definition* is a list of names, each optionally preceded by reserved word **not**, also written in the syntax of a positional_array_aggregate. 3/5

For a nonformal private type, nonformal private extension, or full type that does not have a partial view, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1): 4/5

Stable_Properties 5/5

This aspect shall be specified by a type property aspect definition; each name shall statically denote a single property function of the type. This aspect defines the *specific stable property functions* of the associated type.

Stable_Properties'Class 6/5

This aspect shall be specified by a type property aspect definition; each name shall statically denote a single property function of the type. This aspect defines the *class-wide stable property functions* of the associated type. Unlike most class-wide aspects, Stable_Properties'Class is not inherited by descendant types and subprograms, but the enhanced class-wide postconditions are inherited in the normal manner.

The specific and class-wide stable properties of a type together comprise the stable properties of the type. 7/5

For a primitive subprogram, the following language-defined aspects may be specified with an aspect_specification (see 13.1.1): 8/5

Stable_Properties 9/5

This aspect shall be specified by a subprogram property aspect definition; each name shall statically denote a single property function of a type for which the associated subprogram is primitive.

Stable_Properties'Class 10/5

This aspect shall be specified by a subprogram property aspect definition; each name shall statically denote a single property function of a tagged type for which the associated subprogram is primitive. Unlike most class-wide aspects, Stable_Properties'Class is not inherited by descendant subprograms, but the enhanced class-wide postconditions are inherited in the normal manner.

Legality Rules

A stable property function of a type *T* shall have a nonlimited return type and shall be: 11/5

- a primitive function with a single parameter of mode **in** of type *T*; or 12/5
- a function that is declared immediately within the declarative region in which an ancestor type of *T* is declared and has a single parameter of mode **in** of a class-wide type that covers *T*. 13/5

In a subprogram property aspect definition for a subprogram *S*: 14/5

- all or none of the items shall be preceded by **not**; 15/5

16/5 • any property functions mentioned after **not** shall be a stable property function of a type for which *S* is primitive.

17/5 If a subprogram_renaming_declaration declares a primitive subprogram of a type *T*, then the renamed callable entity shall also be a primitive subprogram of type *T* and the two primitive subprograms shall have the same specific stable property functions and the same class-wide stable property functions (see below).

Static Semantics

18/5 For a primitive subprogram *S* of a type *T*, the specific stable property functions of *S* for type *T* are:

19/5 • if *S* has an aspect Stable_Properties specified that does not include **not**, those functions denoted in the aspect Stable_Properties for *S* that have a parameter of *T* or *T*'Class;

20/5 • if *S* has an aspect Stable_Properties specified that includes **not**, those functions denoted in the aspect Stable_Properties for *T*, excluding those denoted in the aspect Stable_Properties for *S*;

21/5 • if *S* does not have an aspect Stable_Properties, those functions denoted in the aspect Stable_Properties for *T*, if any.

22/5 A similar definition applies for class-wide stable property functions by replacing aspect Stable_Properties with aspect Stable_Properties'Class in the above definition.

23/5 The *explicit* specific postcondition expression for a subprogram *S* is the expression directly specified for *S* with the Post aspect. Similarly, the *explicit* class-wide postcondition expression for a subprogram *S* is the expression directly specified for *S* with the Post'Class aspect.

24/5 For a primitive subprogram *S* of a type *T* that has a parameter *P* of type *T*, the parameter is *excluded from stable property checks* if:

25/5 • *S* is a stable property function of *T*;

26/5 • *P* has mode **out**;

27/5 • the Global aspect of *S* is **null**, and *P* has mode **in** and the mode is not overridden by a global aspect.

28/5 For every primitive subprogram *S* of a type *T* that is not an abstract subprogram or null procedure, the specific postcondition expression of *S* is modified to include expressions of the form `F(P) = F(P)'Old`, all **and**ed with each other and any explicit specific postcondition expression, with one such equality included for each specific stable property function *F* of *S* for type *T* that does not occur in the explicit specific postcondition expression of *S*, and *P* is each parameter of *S* that has type *T* and is not excluded from stable property checks. The resulting specific postcondition expression of *S* is used in place of the explicit specific postcondition expression of *S* when interpreting the meaning of the postcondition as defined in 6.1.1.

29/5 For every primitive subprogram *S* of a type *T*, the class-wide postcondition expression of *S* is modified to include expressions of the form `F(P) = F(P)'Old`, all **and**ed with each other and any explicit class-wide postcondition expression, with one such equality included for each class-wide stable property function *F* of *S* for type *T* that does not occur in any class-wide postcondition expression that applies to *S*, and *P* is each parameter of *S* that has type *T* and is not excluded from stable property checks. The resulting class-wide postcondition expression of *S* is used in place of the explicit class-wide postcondition expression of *S* when interpreting the meaning of the postcondition as defined in 6.1.1.

30/5 The equality operation that is used in the aforementioned equality expressions is as described in the case of an individual membership test whose membership_choice is a *choice*_simple_expression (see 4.5.2).

The Post expression additions described above are enabled or disabled depending on the Post assertion policy that is in effect at the point of declaration of the subprogram *S*. A similar rule applies to the Post'Class expression additions. 31/5

NOTE For an example of the use of these aspects, see the Vector container definition in A.18.2. 32/5

7.4 Deferred Constants

Deferred constant declarations may be used to declare constants in the visible part of a package, but with the value of the constant given in the private part. They may also be used to declare constants imported from other languages (see Annex B). 1

Legality Rules

A *deferred constant declaration* is an object_declaration with the reserved word **constant** but no initialization expression. The constant declared by a deferred constant declaration is called a *deferred constant*. Unless the Import aspect (see B.1) is True for a deferred constant declaration, the deferred constant declaration requires a completion, which shall be a full constant declaration (called the *full declaration* of the deferred constant). 2/3

A deferred constant declaration that is completed by a full constant declaration shall occur immediately within the visible part of a package_specification. For this case, the following additional rules apply to the corresponding full declaration: 3

- The full declaration shall occur immediately within the private part of the same package; 4
- The deferred and full constants shall have the same type, or shall have statically matching anonymous access subtypes; 5/2
- If the deferred constant declaration includes a subtype_indication that defines a subtype *S1*, then the subtype_indication in the full declaration shall define a subtype *S2* that is statically compatible with *S1* (see 4.9.1). If *S1* is a constrained subtype, the constraint defined by *S2* shall statically match the constraint defined by *S1*. If the subtype *S1* of the deferred constant is unconstrained, then the full declaration is still allowed to impose a constraint; 6/6
- If the deferred constant declaration includes the reserved word **aliased**, then the full declaration shall also. 7/6
- 7.1/6

A deferred constant declaration for which the Import aspect is True can appear anywhere that an object_declaration is allowed, and has no full constant declaration. 8/5

The completion of a deferred constant declaration shall occur before the constant is frozen (see 13.14). 9/2

Dynamic Semantics

The elaboration of a deferred constant declaration elaborates the subtype_indication, access_definition, or (only allowed in the case of an imported constant) the array_type_definition. 10/3

NOTE The full constant declaration for a deferred constant that is of a given private type or private extension is not allowed before the corresponding full_type_declaration. This is a consequence of the freezing rules for types (see 13.14). 11

Examples

Examples of deferred constant declarations: 12

```
Null_Key : constant Key;        -- see 7.3.1
```
13

14/3
```
CPU_Identifier : constant String(1..8)
   with Import => True, Convention => Assembler, Link_Name => "CPU_ID";
                                -- see B.1
```

7.5 Limited Types

1/2 A limited type is (a view of) a type for which copying (such as for an assignment_statement) is not allowed. A nonlimited type is a (view of a) type for which copying is allowed.

Legality Rules

2/2 If a tagged record type has any limited components, then the reserved word **limited** shall appear in its record_type_definition. If the reserved word **limited** appears in the definition of a derived_type_definition, its parent type and any progenitor interfaces shall be limited.

2.1/5 In the following contexts, an expression of a limited type is permitted only if each of its operative constituents is newly constructed (see 4.4):

2.2/2 • the initialization expression of an object_declaration (see 3.3.1)

2.3/2 • the default_expression of a component_declaration (see 3.8)

2.4/2 • the expression of a record_component_association (see 4.3.1)

2.5/2 • the expression for an ancestor_part of an extension_aggregate (see 4.3.2)

2.6/2 • an expression of a positional_array_aggregate or the expression of an array_component_association (see 4.3.3)

2.7/5 • the *base*_expression of a record_delta_aggregate (see 4.3.4)

2.8/5 • the qualified_expression of an initialized allocator (see 4.8)

2.9/5 • the expression of a return statement (see 6.5)

2.10/5 • the return expression of an expression function (see 6.8)

2.11/5 • the default_expression or actual parameter for a formal object of mode **in** (see 12.4)

Static Semantics

3/3 A view of a type is *limited* if it is one of the following:

4/2 • a type with the reserved word **limited**, **synchronized**, **task**, or **protected** in its definition;

5/3 • a class-wide type whose specific type is limited;

6/2 • a composite type with a limited component;

6.1/3 • an incomplete view;

6.2/2 • a derived type whose parent is limited and is not an interface.

7 Otherwise, the type is nonlimited.

8 There are no predefined equality operators for a limited type.

8.1/3 A type is *immutably limited* if it is one of the following:

8.2/3 • An explicitly limited record type;

8.3/3 • A record extension with the reserved word **limited**;

8.4/3 • A nonformal limited private type that is tagged or has at least one access discriminant with a default_expression;

- A task type, a protected type, or a synchronized interface; 8.5/3
- A type derived from an immutably limited type. 8.6/3

A descendant of a generic formal limited private type is presumed to be immutably limited except within the body of a generic unit or a body declared within the declarative region of a generic unit, if the formal type is declared within the formal part of the generic unit. 8.7/3

NOTE 1 While it is allowed to write initializations of limited objects, such initializations never copy a limited object. The source of such an assignment operation will be an aggregate or function_call, and such aggregates and function_calls will be built directly in the target object (see 7.6). 9/5

Paragraphs 10 through 15 were deleted.

NOTE 2 As illustrated in 7.3.1, an untagged limited type can become nonlimited under certain circumstances. 16

Examples

Example of a package with a limited type: 17

```
package IO_Package is                                                    18
   type File_Name is limited private;

   procedure Open (F : in out File_Name);                                19
   procedure Close(F : in out File_Name);
   procedure Read (F : in File_Name; Item : out Integer);
   procedure Write(F : in File_Name; Item : in  Integer);
private
   type File_Name is
      limited record
         Internal_Name : Integer := 0;
      end record;
end IO_Package;

package body IO_Package is                                               20
   Limit : constant := 200;
   type File_Descriptor is record  ...  end record;
   Directory : array (1 .. Limit) of File_Descriptor;
   ...
   procedure Open (F : in out File_Name) is  ...  end;
   procedure Close(F : in out File_Name) is  ...  end;
   procedure Read (F : in File_Name; Item : out Integer) is ... end;
   procedure Write(F : in File_Name; Item : in  Integer) is ... end;
begin
   ...
end IO_Package;
```

In the example, an outside subprogram making use of IO_Package can obtain a file name by calling Open and later use it in calls to Read and Write. Thus, outside the package, a file name obtained from Open acts as a kind of password; its internal properties (such as containing a numeric value) are not known and no other operations (such as addition or comparison of internal names) can be performed on a file name. Most importantly, clients of the package cannot make copies of objects of type File_Name. 21/5

This example is characteristic of any case where complete control over the operations of a type is desired. Such packages serve a dual purpose. They prevent a user from making use of the internal structure of the type. They also implement the notion of an encapsulated data type where the only operations on the type are those given in the package specification. 22/5

The fact that the full view of File_Name is explicitly declared **limited** means that parameter passing will always be by reference and function results will always be built directly in the result object (see 6.2 and 6.5). 23/5

7.6 Assignment and Finalization

1 Three kinds of actions are fundamental to the manipulation of objects: initialization, finalization, and assignment. Every object is initialized, either explicitly or by default, after being created (for example, by an object_declaration or allocator). Every object is finalized before being destroyed (for example, by leaving a subprogram_body containing an object_declaration, or by a call to an instance of Unchecked_Deallocation). An assignment operation is used as part of assignment_statements, explicit initialization, parameter passing, and other operations.

2 Default definitions for these three fundamental operations are provided by the language, but a *controlled* type gives the user additional control over parts of these operations. In particular, the user can define, for a controlled type, an Initialize procedure which is invoked immediately after the normal default initialization of a controlled object, a Finalize procedure which is invoked immediately before finalization of any of the components of a controlled object, and an Adjust procedure which is invoked as the last step of an assignment to a (nonlimited) controlled object.

Static Semantics

3 The following language-defined library package exists:

4/5
```
package Ada.Finalization
    with Pure, Nonblocking => False is

5/5     type Controlled is abstract tagged private
           with Preelaborable_Initialization;

6/2     procedure Initialize (Object : in out Controlled) is null;
        procedure Adjust     (Object : in out Controlled) is null;
        procedure Finalize   (Object : in out Controlled) is null;

7/5     type Limited_Controlled is abstract tagged limited private
           with Preelaborable_Initialization;

8/2     procedure Initialize (Object : in out Limited_Controlled) is null;
        procedure Finalize   (Object : in out Limited_Controlled) is null;
private
        ... -- not specified by the language
end Ada.Finalization;
```

9/2 A controlled type is a descendant of Controlled or Limited_Controlled. The predefined "=" operator of type Controlled always returns True, since this operator is incorporated into the implementation of the predefined equality operator of types derived from Controlled, as explained in 4.5.2. The type Limited_Controlled is like Controlled, except that it is limited and it lacks the primitive subprogram Adjust.

9.1/2 A type is said to *need finalization* if:

9.2/2 - it is a controlled type, a task type or a protected type; or

9.3/3 - it has a component whose type needs finalization; or

9.4/3 - it is a class-wide type; or

9.5/3 - it is a partial view whose full view needs finalization; or

9.6/2 - it is one of a number of language-defined types that are explicitly defined to need finalization.

Dynamic Semantics

10/2 During the elaboration or evaluation of a construct that causes an object to be initialized by default, for every controlled subcomponent of the object that is not assigned an initial value (as defined in 3.3.1),

Initialize is called on that subcomponent. Similarly, if the object that is initialized by default as a whole is controlled, Initialize is called on the object.

For an extension_aggregate whose ancestor_part is a subtype_mark denoting a controlled subtype, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract. 11/2

Initialize and other initialization operations are done in an arbitrary order, except as follows. Initialize is applied to an object after initialization of its subcomponents, if any (including both implicit initialization and Initialize calls). If an object has a component with an access discriminant constrained by a per-object expression, Initialize is applied to this component after any components that do not have such discriminants. For an object with several components with such a discriminant, Initialize is applied to them in order of their component_declarations. For an allocator, any task activations follow all calls on Initialize. 12

When a target object with any controlled parts is assigned a value, either when created or in a subsequent assignment_statement, the *assignment operation* proceeds as follows: 13

- The value of the target becomes the assigned value. 14
- The value of the target is *adjusted.* 15

To adjust the value of a composite object, the values of the components of the object are first adjusted in an arbitrary order, and then, if the object is nonlimited controlled, Adjust is called. Adjusting the value of an elementary object has no effect, nor does adjusting the value of a composite object with no controlled parts. 16/3

For an assignment_statement, after the name and expression have been evaluated, and any conversion (including constraint checking) has been done, an anonymous object is created, and the value is assigned into it; that is, the assignment operation is applied. (Assignment includes value adjustment.) The target of the assignment_statement is then finalized. The value of the anonymous object is then assigned into the target of the assignment_statement. Finally, the anonymous object is finalized. As explained below, the implementation may eliminate the intermediate anonymous object, so this description subsumes the one given in 5.2, “Assignment Statements”. 17

When a function call or aggregate is used to initialize an object, the result of the function call or aggregate is an anonymous object, which is assigned into the newly-created object. For such an assignment, the anonymous object may be *built in place*, in which case the assignment does not involve any copying. Under certain circumstances, the anonymous object is required to be built in place. In particular: 17.1/5

- If the full type of any part of the object is immutably limited, the anonymous object is built in place. 17.2/3
- In the case of an aggregate, if the full type of any part of the newly-created object is controlled, the anonymous object is built in place. 17.3/3
- In other cases, it is unspecified whether the anonymous object is built in place. 17.4/3

Notwithstanding what this document says elsewhere, if an object is built in place: 17.5/3

- Upon successful completion of the return statement or aggregate, the anonymous object *mutates into* the newly-created object; that is, the anonymous object ceases to exist, and the newly-created object appears in its place. 17.6/3
- Finalization is not performed on the anonymous object. 17.7/3
- Adjustment is not performed on the newly-created object. 17.8/3

17.9/3 • All access values that designate parts of the anonymous object now designate the corresponding parts of the newly-created object.

17.10/3 • All renamings of parts of the anonymous object now denote views of the corresponding parts of the newly-created object.

17.11/3 • Coextensions of the anonymous object become coextensions of the newly-created object.

Implementation Permissions

17.13/6 If the result of a function call is used (in its entirety) to directly initialize an object, but the result is not built in place, the anonymous object representing the result of the function call may be finalized and become nonexistent immediately after the initialization of the newly-created object is complete.

18/3 An implementation is allowed to relax the above rules for assignment_statements in the following ways:

19/5 • If an object is assigned the value of that same object, the implementation may omit the entire assignment.

20/3 • For assignment of a noncontrolled type, the implementation may finalize and assign each component of the variable separately (rather than finalizing the entire variable and assigning the entire new value) unless a discriminant of the variable is changed by the assignment.

21/5 • The implementation may avoid creating an anonymous object if the value being assigned is the result of evaluating a name denoting an object (the source object) whose storage cannot overlap with the target. If the source object can overlap with the target object, then the implementation can avoid the need for an intermediary anonymous object by exercising one of the above permissions and perform the assignment one component at a time (for an overlapping array assignment), or not at all (for an assignment where the target and the source of the assignment are the same object).

22/2 Furthermore, an implementation is permitted to omit implicit Initialize, Adjust, and Finalize calls and associated assignment operations on an object of a nonlimited controlled type provided that:

23/2 • any omitted Initialize call is not a call on a user-defined Initialize procedure, and

24/2 • any usage of the value of the object after the implicit Initialize or Adjust call and before any subsequent Finalize call on the object does not change the external effect of the program, and

25/2 • after the omission of such calls and operations, any execution of the program that executes an Initialize or Adjust call on an object or initializes an object by an aggregate will also later execute a Finalize call on the object and will always do so prior to assigning a new value to the object, and

26/2 • the assignment operations associated with omitted Adjust calls are also omitted.

27/2 This permission applies to Adjust and Finalize calls even if the implicit calls have additional external effects.

7.6.1 Completion and Finalization

This subclause defines *completion* and *leaving* of the execution of constructs and entities. A *master* is the execution of a construct that includes finalization of local objects after it is complete (and after waiting for any local tasks — see 9.3), but before leaving. Other constructs and entities are left immediately upon completion. 1

Dynamic Semantics

The execution of a construct or entity is *complete* when the end of that execution has been reached, or when a transfer of control (see 5.1) causes it to be abandoned. Completion due to reaching the end of execution, or due to the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement or of the selection of a terminate_alternative is *normal completion*. Completion is *abnormal* otherwise — when control is transferred out of a construct due to abort or the raising of an exception. 2/2

After execution of a construct or entity is complete, it is *left*, meaning that execution continues with the next action, as defined for the execution that is taking place. Leaving an execution happens immediately after its completion, except in the case of the execution of a *master construct*: a body other than a package_body; a statement; an expression, function_call, or range that is not part of an enclosing expression, function_call, range, or simple_statement other than a simple_return_statement; or an expression or <> that appears in an iterative context (see 4.4). The term *master* by itself refers to the execution of a master construct. A master is finalized after it is complete, and before it is left. 3/6

For the *finalization* of a master, dependent tasks are first awaited, as explained in 9.3. Then each object whose accessibility level is the same as that of the master is finalized if the object was successfully initialized and still exists. These actions are performed whether the master is left by reaching the last statement or via a transfer of control. When a transfer of control causes completion of an execution, each included master is finalized in order, from innermost outward. 4

For the *finalization* of an object: 5

- If the full type of the object is an elementary type, finalization has no effect; 6/3
- If the full type of the object is a tagged type, and the tag of the object identifies a controlled type, the Finalize procedure of that controlled type is called; 7/3
- If the full type of the object is a protected type, or if the full type of the object is a tagged type and the tag of the object identifies a protected type, the actions defined in 9.4 are performed; 8/3
- If the full type of the object is a composite type, then after performing the above actions, if any, every component of the object is finalized in an arbitrary order, except as follows: if the object has a component with an access discriminant constrained by a per-object expression, this component is finalized before any components that do not have such discriminants; for an object with several components with such a discriminant, they are finalized in the reverse of the order of their component_declarations; 9/3
- If the object has coextensions (see 3.10.2), each coextension is finalized after the object whose access discriminant designates it. 9.1/2

Immediately before an instance of Unchecked_Deallocation reclaims the storage of an object, the object is finalized. If an instance of Unchecked_Deallocation is never applied to an object created by an allocator, the object will still exist when the corresponding master completes, and it will be finalized then. 10

11/6 The finalization of a master performs finalization of objects (other than components) created by declarations in the master in the reverse order of their creation. The order in which finalization of other objects in the master are finalized is unspecified with respect to any other objects of the same master. After the finalization of a master is complete, the objects finalized as part of its finalization cease to *exist*, as do any types and subtypes defined and created within the master.

11.1/3 Each nonderived access type *T* has an associated *collection*, which is the set of objects created by allocators of *T*, or of types derived from *T*. Unchecked_Deallocation removes an object from its collection. Finalization of a collection consists of finalization of each object in the collection, in an arbitrary order. The collection of an access type is an object implicitly declared at the following place:

11.2/3 • For a named access type, the first freezing point (see 13.14) of the type.

11.3/6 • For the type of an access parameter, the evaluation of the call that contains the allocator.

11.4/6 • For the type of an access result, within the master of the function call (see 3.10.2).

11.5/3 • For any other anonymous access type, the first freezing point of the innermost enclosing declaration.

12/2 The target of an assignment_statement is finalized before copying in the new value, as explained in 7.6.

12.1/6 When an assignment changes the value of at least one discriminant of a target object, every discriminant-dependent component of the previous value ceases to exist, and every discriminant-dependent component of the new value of the object is newly created.

13/3 The master of an object is the master enclosing its creation whose accessibility level (see 3.10.2) is equal to that of the object, except in the case of an anonymous object representing the result of an aggregate or function call. If such an anonymous object is part of the result of evaluating the actual parameter expression for an explicitly aliased parameter of a function call, the master of the object is the innermost master enclosing the evaluation of the aggregate or function call, excluding the aggregate or function call itself. Otherwise, the master of such an anonymous object is the innermost master enclosing the evaluation of the aggregate or function call, which may be the aggregate or function call itself.

13.1/6 In the case of an expression or <> that is a master construct, finalization of the associated master occurs after completing evaluation of the expression or <> and all use of any objects associated with the expression or <>, prior to starting the execution of any subsequent construct or iteration.

Bounded (Run-Time) Errors

14/1 It is a bounded error for a call on Finalize or Adjust that occurs as part of object finalization or assignment to propagate an exception. The possible consequences depend on what action invoked the Finalize or Adjust operation:

15 • For a Finalize invoked as part of an assignment_statement, Program_Error is raised at that point.

16/5 • For an Adjust invoked as part of assignment operations other than those invoked as part of an assignment_statement, some of the adjustments due to be performed can be performed, and then Program_Error is raised. During its propagation, finalization may be applied to objects whose Adjust failed. For an Adjust invoked as part of an assignment_statement, any other adjustments due to be performed are performed, and then Program_Error is raised.

17 • For a Finalize invoked as part of a call on an instance of Unchecked_Deallocation, any other finalizations due to be performed are performed, and then Program_Error is raised.

17.1/3 • *This paragraph was deleted.*

- For a Finalize invoked due to reaching the end of the execution of a master, any other finalizations associated with the master are performed, and Program_Error is raised immediately after leaving the master. 17.2/1
- For a Finalize invoked by the transfer of control of an exit_statement, return statement, goto_statement, or requeue_statement, Program_Error is raised no earlier than after the finalization of the master being finalized when the exception occurred, and no later than the point where normal execution would have continued. Any other finalizations due to be performed up to that point are performed before raising Program_Error. 18/2
- For a Finalize invoked by a transfer of control that is due to raising an exception, any other finalizations due to be performed for the same master are performed; Program_Error is raised immediately after leaving the master. 19
- For a Finalize invoked by a transfer of control due to an abort or selection of a terminate alternative, the exception is ignored; any other finalizations due to be performed are performed. 20

Implementation Permissions

If the execution of an allocator propagates an exception, any parts of the allocated object that were successfully initialized may be finalized as part of the finalization of the innermost master enclosing the allocator. 20.1/3

The implementation may finalize objects created by allocators for an access type whose storage pool supports subpools (see 13.11.4) as if the objects were created (in an arbitrary order) at the point where the storage pool was elaborated instead of at the first freezing point of the access type. 20.2/3

NOTE 1 The rules of Clause 10 imply that immediately prior to partition termination, Finalize operations are applied to library-level controlled objects (including those created by allocators of library-level access types, except those already finalized). This occurs after waiting for library-level tasks to terminate. 21/5

NOTE 2 A constant is only constant between its initialization and finalization. Both initialization and finalization are allowed to change the value of a constant. 22

NOTE 3 Abort is deferred during certain operations related to controlled types, as explained in 9.8. Those rules prevent an abort from causing a controlled object to be left in an ill-defined state. 23

NOTE 4 The Finalize procedure is called upon finalization of a controlled object, even if Finalize was called earlier, either explicitly or as part of an assignment; hence, if a controlled type is visibly controlled (implying that its Finalize primitive is directly callable), or is nonlimited (implying that assignment is allowed), its Finalize procedure is ideally designed to have no ill effect if it is applied a second time to the same object. 24

8 Visibility Rules

The rules defining the scope of declarations and the rules defining which identifiers, character_literals, and operator_symbols are visible at (or from) various places in the text of the program are described in this clause. The formulation of these rules uses the notion of a declarative region. 1/3

As explained in Clause 3, a declaration declares a view of an entity and associates a defining name with that view. The view comprises an identification of the viewed entity, and possibly additional properties. A usage name denotes a declaration. It also denotes the view declared by that declaration, and denotes the entity of that view. Thus, two different usage names can denote two different views of the same entity; in this case they denote the same entity. 2/5

8.1 Declarative Region

Static Semantics

For each of the following constructs, there is a portion of the program text called its *declarative region*: 1/6

- any declaration, other than that of an enumeration type, that is not a completion of a previous declaration; 2
- an access_definition; 2.1/4
- an iterated_component_association; 2.2/5
- an iterated_element_association; 2.3/5
- a quantified_expression; 2.4/5
- a declare_expression; 2.5/5
- a block_statement; 3
- a loop_statement; 4
- *This paragraph was deleted.* 4.1/5
- an extended_return_statement; 4.2/3
- an accept_statement; 5
- an exception_handler. 6

The declarative region includes the text of the construct together with additional text determined (recursively), as follows: 7

- If a declaration is included, so is its completion, if any. 8
- If the declaration of a library unit (including Standard — see 10.1.1) is included, so are the declarations of any child units (and their completions, by the previous rule). The child declarations occur after the declaration. 9
- If a body_stub is included, so is the corresponding subunit. 10
- If a type_declaration is included, then so is a corresponding record_representation_clause, if any. 11

The declarative region of a declaration is also called the *declarative region* of any view or entity declared by the declaration. 12

13/6 *Nested* declarations can occur within a declarative region. A declaration occurs *immediately within* a declarative region if this region is the innermost declarative region that encloses the declaration (the *immediately enclosing* declarative region), not counting the declarative region (if any) associated with the declaration itself.

14 A declaration is *local* to a declarative region if the declaration occurs immediately within the declarative region. An entity is *local* to a declarative region if the entity is declared by a declaration that is local to the declarative region.

15 A declaration is *global* to a declarative region if the declaration occurs immediately within another declarative region that encloses the declarative region. An entity is *global* to a declarative region if the entity is declared by a declaration that is global to the declarative region.

16 NOTE 1 The children of a parent library unit are inside the parent's declarative region, even though they do not occur inside the parent's declaration or body. This implies that one can use (for example) "P.Q" to refer to a child of P whose defining name is Q, and that after "**use** P;" Q can refer (directly) to that child.

17 NOTE 2 As explained above and in 10.1.1, “Compilation Units - Library Units”, all library units are descendants of Standard, and so are contained in the declarative region of Standard. They are *not* inside the declaration or body of Standard, but they *are* inside its declarative region.

18/5 NOTE 3 For a declarative region that comes in multiple parts, the text of the declarative region does not include any of the text that appears between the parts. Thus, when a portion of a declarative region is said to extend from one place to another in the declarative region, the portion does not contain any of the text that appears between the parts of the declarative region.

8.2 Scope of Declarations

1 For each declaration, the language rules define a certain portion of the program text called the *scope* of the declaration. The scope of a declaration is also called the scope of any view or entity declared by the declaration. Within the scope of an entity, and only there, there are places where it is legal to refer to the declared entity. These places are defined by the rules of visibility and overloading.

Static Semantics

2 The *immediate scope* of a declaration is a portion of the declarative region immediately enclosing the declaration. The immediate scope starts at the beginning of the declaration, except in the case of an overloadable declaration, in which case the immediate scope starts just after the place where the profile of the callable entity is determined (which is at the end of the _specification for the callable entity, or at the end of the generic_instantiation if an instance). The immediate scope extends to the end of the declarative region, with the following exceptions:

3 • The immediate scope of a library_item includes only its semantic dependents.

4 • The immediate scope of a declaration in the private part of a library unit does not include the visible part of any public descendant of that library unit.

5 The *visible part* of (a view of) an entity is a portion of the text of its declaration containing declarations that are visible from outside. The *private part* of (a view of) an entity that has a visible part contains all declarations within the declaration of (the view of) the entity, except those in the visible part; these are not visible from outside. Visible and private parts are defined only for these kinds of entities: callable entities, other program units, and composite types.

6 • The visible part of a view of a callable entity is its profile.

7 • The visible part of a composite type other than a task or protected type consists of the declarations of all components declared (explicitly or implicitly) within the type_declaration.

- The visible part of a generic unit includes the generic_formal_part. For a generic package, it also includes the first list of basic_declarative_items of the package_specification. For a generic subprogram, it also includes the profile. 8
- The visible part of a package, task unit, or protected unit consists of declarations in the program unit's declaration other than those following the reserved word **private**, if any; see 7.1 and 12.7 for packages, 9.1 for task units, and 9.4 for protected units. 9

The scope of a declaration always contains the immediate scope of the declaration. In addition, for a given declaration that occurs immediately within the visible part of an outer declaration, or is a public child of an outer declaration, the scope of the given declaration extends to the end of the scope of the outer declaration, except that the scope of a library_item includes only its semantic dependents. 10

The scope of an attribute_definition_clause is identical to the scope of a declaration that would occur at the point of the attribute_definition_clause. The scope of an aspect_specification is identical to the scope of the associated declaration. 10.1/3

The immediate scope of a declaration is also the immediate scope of the entity or view declared by the declaration. Similarly, the scope of a declaration is also the scope of the entity or view declared by the declaration. 11

The immediate scope of a pragma that is not used as a configuration pragma is defined to be the region extending from immediately after the pragma to the end of the declarative region immediately enclosing the pragma. 12/4

NOTE There are notations for denoting visible declarations that are not directly visible. For example, parameter_specifications are in the visible part of a subprogram_declaration so that they can be used in named-notation calls appearing outside the called subprogram. For another example, declarations of the visible part of a package can be denoted by expanded names appearing outside the package, and can be made directly visible by a use_clause. 13/3

8.3 Visibility

The *visibility rules*, given below, determine which declarations are visible and directly visible at each place within a program. The visibility rules apply to both explicit and implicit declarations. 1

Static Semantics

A declaration is defined to be *directly visible* at places where a name consisting of only an identifier or operator_symbol is sufficient to denote the declaration; that is, no selected_component notation or special context (such as preceding => in a named association) is necessary to denote the declaration. A declaration is defined to be *visible* wherever it is directly visible, as well as at other places where some name (such as a selected_component) can denote the declaration. 2

The syntactic category direct_name is used to indicate contexts where direct visibility is required. The syntactic category selector_name is used to indicate contexts where visibility, but not direct visibility, is required. 3

There are two kinds of direct visibility: *immediate visibility* and *use-visibility*. A declaration is immediately visible at a place if it is directly visible because the place is within its immediate scope. A declaration is use-visible if it is directly visible because of a use_clause (see 8.4). Both conditions can apply. 4

A declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope. Where *hidden from all visibility*, it is not visible at all (neither using a direct_name nor a 5

selector_name). Where *hidden from direct visibility*, only direct visibility is lost; visibility using a selector_name is still possible.

6 Two or more declarations are *overloaded* if they all have the same defining name and there is a place where they are all directly visible.

7 The declarations of callable entities (including enumeration literals) are *overloadable*, meaning that overloading is allowed for them.

8 Two declarations are *homographs* if they have the same defining name, and, if both are overloadable, their profiles are type conformant. An inner declaration hides any outer homograph from direct visibility.

9/1 Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below). The only declarations that are *overridable* are the implicit declarations for predefined operators and inherited primitive subprograms. A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:

10/1 - A declaration that is not overridable overrides one that is overridable, regardless of which declaration occurs first;

11 - The implicit declaration of an inherited operator overrides that of a predefined operator;

12 - An implicit declaration of an inherited subprogram overrides a previous implicit declaration of an inherited subprogram.

12.1/2 - If two or more homographs are implicitly declared at the same place:

12.2/2 - If at least one is a subprogram that is neither a null procedure nor an abstract subprogram, and does not require overriding (see 3.9.3), then they override those that are null procedures, abstract subprograms, or require overriding. If more than one such homograph remains that is not thus overridden, then they are all hidden from all visibility.

12.3/5 - Otherwise (all are null procedures, abstract subprograms, or require overriding), then any null procedure overrides all abstract subprograms and all subprograms that require overriding; if more than one such homograph remains that is not thus overridden, then if the profiles of the remaining homographs are all fully conformant with one another, one is chosen arbitrarily; if not, they are all hidden from all visibility.

13 - For an implicit declaration of a primitive subprogram in a generic unit, there is a copy of this declaration in an instance. However, a whole new set of primitive subprograms is implicitly declared for each type declared within the visible part of the instance. These new declarations occur immediately after the type declaration, and override the copied ones. The copied ones can be called only from within the instance; the new ones can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one.

14 A declaration is visible within its scope, except where hidden from all visibility, as follows:

15 - An overridden declaration is hidden from all visibility within the scope of the overriding declaration.

16 - A declaration is hidden from all visibility until the end of the declaration, except:

17 - For a record type or record extension, the declaration is hidden from all visibility only until the reserved word **record**;

18/3 - For a package_declaration, generic_package_declaration, subprogram_body, or expression_function_declaration, the declaration is hidden from all visibility only until the reserved word **is** of the declaration;

- For a task declaration or protected declaration, the declaration is hidden from all visibility only until the reserved word **with** of the declaration if there is one, or the reserved word **is** of the declaration if there is no **with**. 18.1/2

- If the completion of a declaration is a declaration, then within the scope of the completion, the first declaration is hidden from all visibility. Similarly, a discriminant_specification or parameter_specification is hidden within the scope of a corresponding discriminant_specification or parameter_specification of a corresponding completion, or of a corresponding accept_statement. 19

- The declaration of a library unit (including a library_unit_renaming_declaration) is hidden from all visibility at places outside its declarative region that are not within the scope of a nonlimited_with_clause that mentions it. The limited view of a library package is hidden from all visibility at places that are not within the scope of a limited_with_clause that mentions it; in addition, the limited view is hidden from all visibility within the declarative region of the package, as well as within the scope of any nonlimited_with_clause that mentions the package. Where the declaration of the limited view of a package is visible, any name that denotes the package denotes the limited view, including those provided by a package renaming. 20/2

- For each declaration or renaming of a generic unit as a child of some parent generic package, there is a corresponding declaration nested immediately within each instance of the parent. Such a nested declaration is hidden from all visibility except at places that are within the scope of a with_clause that mentions the child. 20.1/2

A declaration with a defining_identifier or defining_operator_symbol is immediately visible (and hence directly visible) within its immediate scope except where hidden from direct visibility, as follows: 21

- A declaration is hidden from direct visibility within the immediate scope of a homograph of the declaration, if the homograph occurs within an inner declarative region; 22

- A declaration is also hidden from direct visibility where hidden from all visibility. 23

An attribute_definition_clause or an aspect_specification is *visible* everywhere within its scope. 23.1/3

Name Resolution Rules

A direct_name shall resolve to denote a directly visible declaration whose defining name is the same as the direct_name. A selector_name shall resolve to denote a visible declaration whose defining name is the same as the selector_name. 24

These rules on visibility and direct visibility do not apply in a context_clause, a parent_unit_name, or a pragma that appears at the place of a compilation_unit. For those contexts, see the rules in 10.1.6, “Environment-Level Visibility Rules”. 25

Legality Rules

A nonoverridable declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the nonoverridable declaration. This rule also applies to dispatching operations declared in the visible part of an instance of a generic unit. However, it does not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant. 26/6

A type extension is illegal if somewhere within its immediate scope it has two visible components with the same name. The context_clause for a compilation unit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is visible at the place of the compilation unit, 26.1/6

and the homograph and the mentioned library unit are both declared immediately within the same declarative region.

27 NOTE 1 Visibility for compilation units follows from the definition of the environment in 10.1.4, except that it is necessary to apply a with_clause to obtain visibility to a library_unit_declaration or library_unit_renaming_declaration.

28 NOTE 2 In addition to the visibility rules given above, the meaning of the occurrence of a direct_name or selector_name at a given place in the text can depend on the overloading rules (see 8.6).

29 NOTE 3 Not all contexts where an identifier, character_literal, or operator_symbol are allowed require visibility of a corresponding declaration. Contexts where visibility is not required are identified by using one of these three syntactic categories directly in a syntax rule, rather than using direct_name or selector_name.

8.3.1 Overriding Indicators

1/2 An overriding_indicator is used to declare that an operation is intended to override (or not override) an inherited operation.

Syntax

2/2 overriding_indicator ::= [**not**] **overriding**

Legality Rules

3/3 If an abstract_subprogram_declaration, null_procedure_declaration, expression_function_declaration, subprogram_body, subprogram_body_stub, subprogram_renaming_declaration, generic_instantiation of a subprogram, or subprogram_declaration other than a protected subprogram has an overriding_indicator, then:

4/2 - the operation shall be a primitive operation for some type;

5/2 - if the overriding_indicator is **overriding**, then the operation shall override a homograph at the place of the declaration or body;

6/2 - if the overriding_indicator is **not overriding**, then the operation shall not override any homograph (at any place).

7/2 In addition to the places where Legality Rules normally apply, these rules also apply in the private part of an instance of a generic unit.

8/2 NOTE Rules for overriding_indicators of task and protected entries and of protected subprograms are found in 9.5.2 and 9.4, respectively.

Examples

9/5 *Example of use of an overriding indicator when declaring a security queue derived from the Queue interface of 3.9.4:*

```
10/2  type Security_Queue is new Queue with record ...;
11/2  overriding
      procedure Append(Q : in out Security_Queue; Person : in Person_Name);
12/5  overriding
      procedure Remove_First(Q : in out Security_Queue; Person : out Person_Name);
13/2  overriding
      function Cur_Count(Q : in Security_Queue) return Natural;
14/2  overriding
      function Max_Count(Q : in Security_Queue) return Natural;
15/2  not overriding
      procedure Arrest(Q : in out Security_Queue; Person : in Person_Name);
```

The first four subprogram declarations guarantee that these subprograms will override the four subprograms inherited from the Queue interface. A misspelling in one of these subprograms will be detected at compile time by the implementation. Conversely, the declaration of Arrest guarantees that this is a new operation. 16/5

8.4 Use Clauses

A use_package_clause achieves direct visibility of declarations that appear in the visible part of a package; a use_type_clause achieves direct visibility of the primitive operators of a type. 1

Syntax

use_clause ::= use_package_clause | use_type_clause 2

use_package_clause ::= **use** *package_*name {, *package_*name}; 3

use_type_clause ::= **use** [**all**] **type** subtype_mark {, subtype_mark}; 4/3

Legality Rules

A *package_*name of a use_package_clause shall denote a nonlimited view of a package. 5/2

Static Semantics

For each use_clause, there is a certain region of text called the *scope* of the use_clause. For a use_clause within a context_clause of a library_unit_declaration or library_unit_renaming_declaration, the scope is the entire declarative region of the declaration. For a use_clause within a context_clause of a body, the scope is the entire body and any subunits (including multiply nested subunits). The scope does not include context_clauses themselves. 6

For a use_clause immediately within a declarative region, the scope is the portion of the declarative region starting just after the use_clause and extending to the end of the declarative region. However, the scope of a use_clause in the private part of a library unit does not include the visible part of any public descendant of that library unit. 7

A package is *named* in a use_package_clause if it is denoted by a *package_*name of that clause. A type is *named* in a use_type_clause if it is determined by a subtype_mark of that clause. 7.1/2

For each package named in a use_package_clause whose scope encloses a place, each declaration that occurs immediately within the declarative region of the package is *potentially use-visible* at this place if the declaration is visible at this place. For each type *T* or *T*'Class named in a use_type_clause whose scope encloses a place, the declaration of each primitive operator of type *T* is potentially use-visible at this place if its declaration is visible at this place. If a use_type_clause whose scope encloses a place includes the reserved word **all**, then the following entities are also potentially use-visible at this place if the declaration of the entity is visible at this place: 8/3

- Each primitive subprogram of *T* including each enumeration literal (if any); 8.1/3
- Each subprogram that is declared immediately within the declarative region in which an ancestor type of *T* is declared and that operates on a class-wide type that covers *T*. 8.2/3

Certain implicit declarations may become potentially use-visible in certain contexts as described in 12.6. 8.3/3

A declaration is *use-visible* if it is potentially use-visible, except in these naming-conflict cases: 9

- A potentially use-visible declaration is not use-visible if the place considered is within the immediate scope of a homograph of the declaration. 10

11 • Potentially use-visible declarations that have the same identifier are not use-visible unless each of them is an overloadable declaration.

Dynamic Semantics

12 The elaboration of a use_clause has no effect.

Examples

13 *Example of a use clause in a context clause:*

14
```
with Ada.Calendar; use Ada;
```

15 *Example of a use type clause:*

16
```
use type Rational_Numbers.Rational; -- see 7.1
Two_Thirds: Rational_Numbers.Rational := 2/3;
```

8.5 Renaming Declarations

1 A renaming_declaration declares another name for an entity, such as an object, exception, package, subprogram, entry, or generic unit. Alternatively, a subprogram_renaming_declaration can be the completion of a previous subprogram_declaration.

Syntax

2
```
renaming_declaration ::=
    object_renaming_declaration
  | exception_renaming_declaration
  | package_renaming_declaration
  | subprogram_renaming_declaration
  | generic_renaming_declaration
```

Dynamic Semantics

3 The elaboration of a renaming_declaration evaluates the name that follows the reserved word **renames** and thereby determines the view and entity denoted by this name (the *renamed view* and *renamed entity*). A name that denotes the renaming_declaration denotes (a new view of) the renamed entity.

4/5 NOTE 1 Renaming can be used to resolve name conflicts and to act as a shorthand. Renaming with a different identifier or operator_symbol does not hide the old name; the new name and the old name can be visible at different places.

5/5 *This paragraph was deleted.*

6 NOTE 2 A subtype defined without any additional constraint can be used to achieve the effect of renaming another subtype (including a task or protected subtype) as in

7
```
subtype Mode is Ada.Text_IO.File_Mode;
```

8.5.1 Object Renaming Declarations

1/5 An object_renaming_declaration is used to rename an object or value.

Syntax

2/5
```
object_renaming_declaration ::=
   defining_identifier [: [null_exclusion] subtype_mark] renames object_name
      [aspect_specification];
 | defining_identifier : access_definition renames object_name
      [aspect_specification];
```

Name Resolution Rules

The type of the *object_*name shall resolve to the type determined by the subtype_mark, if present. If no subtype_mark or access_definition is present, the expected type of the *object_*name is any type. 3/5

In the case where the type is defined by an access_definition, the type of the *object_*name shall resolve to an anonymous access type. If the anonymous access type is an access-to-object type, the type of the *object_*name shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the *object_*name shall have a designated profile that is type conformant with that of the access_definition. 3.1/5

Legality Rules

The renamed entity shall be an object or value. 4/5

In the case where the type is defined by an access_definition, the type of the renamed entity and the type defined by the access_definition: 4.1/5

- shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or 4.2/2
- shall both be access-to-subprogram types with subtype conformant designated profiles. 4.3/2

For an object_renaming_declaration with a null_exclusion or an access_definition that has a null_exclusion, the subtype of the *object_*name shall exclude null. In addition, if the object_renaming_declaration occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of generic unit *G*, then: 4.4/5

- if the *object_*name statically denotes a generic formal object of mode **in out** of *G*, then the declaration of that object shall have a null_exclusion; 4.5/5
- if the *object_*name statically denotes a call of a generic formal function of *G*, then the declaration of the result of that function shall have a null_exclusion. 4.6/5

In the case where the *object_*name is a qualified_expression with a nominal subtype *S* and whose expression is a name that denotes an object *Q*: 4.7/5

- if *S* is an elementary subtype, then: 4.8/5
  - *Q* shall be a constant other than a dereference of an access type; or 4.9/5
  - the nominal subtype of *Q* shall be statically compatible with *S*; or 4.10/5
  - *S* shall statically match the base subtype of its type if scalar, or the first subtype of its type if an access type. 4.11/5
- if *S* is a composite subtype, then *Q* shall be known to be constrained or *S* shall statically match the first subtype of its type. 4.12/5

The renamed entity shall not be a subcomponent that depends on discriminants of an object whose nominal subtype is unconstrained unless the object is known to be constrained. A slice of an array shall not be renamed if this restriction disallows renaming of the array. 5/5

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 5.1/5

Static Semantics

An object_renaming_declaration declares a new view of the renamed entity whose properties are identical to those of the renamed view. Thus, the properties of the renamed entity are not affected by the renaming_declaration. In particular, its nominal subtype, whether it is a value or an object, its value if it is 6/5

an object, and whether or not it is a constant, are unaffected; similarly, the constraints and other properties of its nominal subtype are not affected by renaming (any constraint implied by the subtype_mark or access_definition of the object_renaming_declaration is ignored).

Examples

7 *Example of renaming an object:*

8
```
declare
   L : Person renames Leftmost_Person; -- see 3.10.1
begin
   L.Age := L.Age + 1;
end;
```

9/5 *Example of renaming a value:*

10/5
```
Uno renames One;   -- see 3.3.2
```

8.5.2 Exception Renaming Declarations

1 An exception_renaming_declaration is used to rename an exception.

Syntax

2/3 exception_renaming_declaration ::=
  defining_identifier : **exception renames** *exception_*name
   [aspect_specification];

Legality Rules

3 The renamed entity shall be an exception.

Static Semantics

4 An exception_renaming_declaration declares a new view of the renamed exception.

Examples

5 *Example of renaming an exception:*

6
```
EOF : exception renames Ada.IO_Exceptions.End_Error; -- see A.13
```

8.5.3 Package Renaming Declarations

1 A package_renaming_declaration is used to rename a package.

Syntax

2/3 package_renaming_declaration ::=
  package defining_program_unit_name **renames** *package_*name
   [aspect_specification];

Legality Rules

3 The renamed entity shall be a package.

3.1/2 If the *package_*name of a package_renaming_declaration denotes a limited view of a package *P*, then a name that denotes the package_renaming_declaration shall occur only within the immediate scope of the renaming or the scope of a with_clause that mentions the package *P* or, if *P* is a nested package, the innermost library package enclosing *P*.

Static Semantics

A package_renaming_declaration declares a new view of the renamed package. 4

At places where the declaration of the limited view of the renamed package is visible, a name that denotes the package_renaming_declaration denotes a limited view of the package (see 10.1.1). 4.1/2

Examples

Example of renaming a package: 5

```
package TM renames Table_Manager;
```
6

8.5.4 Subprogram Renaming Declarations

A subprogram_renaming_declaration can serve as the completion of a subprogram_declaration; such a renaming_declaration is called a *renaming-as-body*. A subprogram_renaming_declaration that is not a completion is called a *renaming-as-declaration*, and is used to rename a subprogram (possibly an enumeration literal) or an entry. 1/3

Syntax

subprogram_renaming_declaration ::= 2/3
  [overriding_indicator]
  subprogram_specification **renames** *callable_entity*_name
    [aspect_specification];

Name Resolution Rules

The expected profile for the *callable_entity*_name is the profile given in the subprogram_specification. 3

Legality Rules

The profile of a renaming-as-declaration shall be mode conformant with that of the renamed callable entity. 4/5

For a parameter or result subtype of the subprogram_specification that has an explicit null_exclusion: 4.1/2

- if the *callable_entity*_name statically denotes a generic formal subprogram of a generic unit *G*, and the subprogram_renaming_declaration occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result subtype of the formal subprogram of *G* shall have a null_exclusion; 4.2/5
- otherwise, the subtype of the corresponding parameter or result type of the renamed callable entity shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 4.3/2

The profile of a renaming-as-body shall conform fully to that of the declaration it completes. If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the profile shall be mode conformant with that of the renamed callable entity and the subprogram it declares takes its convention from the renamed subprogram; otherwise, the profile shall be subtype conformant with that of the renamed callable entity and the convention of the renamed subprogram shall not be Intrinsic. A renaming-as-body is illegal if the declaration occurs before the subprogram whose declaration it completes is frozen, and the renaming renames the subprogram itself, through one or more subprogram renaming declarations, none of whose subprograms has been frozen. 5/3

5.1/2 The *callable_entity_*name of a renaming shall not denote a subprogram that requires overriding (see 3.9.3).

5.2/2 The *callable_entity_*name of a renaming-as-body shall not denote an abstract subprogram.

5.3/5 If the *callable_entity_*name of a renaming is a prefixed view, the prefix of that view shall denote an object for which renaming is allowed.

6 A name that denotes a formal parameter of the subprogram_specification is not allowed within the *callable_entity_*name.

Static Semantics

7 A renaming-as-declaration declares a new view of the renamed entity. The profile of this new view takes its subtypes, parameter modes, and calling convention from the original profile of the callable entity, while taking the formal parameter names and default_expressions from the profile given in the subprogram_renaming_declaration. The new view is a function or procedure, never an entry.

Dynamic Semantics

7.1/1 For a call to a subprogram whose body is given as a renaming-as-body, the execution of the renaming-as-body is equivalent to the execution of a subprogram_body that simply calls the renamed subprogram with its formal parameters as the actual parameters and, if it is a function, returns the value of the call.

8/3 For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called. A corresponding rule applies to a call on a renaming of a predefined equality operator for an untagged record type.

Bounded (Run-Time) Errors

8.1/1 If a subprogram directly or indirectly renames itself, then it is a bounded error to call that subprogram. Possible consequences are that Program_Error or Storage_Error is raised, or that the call results in infinite recursion.

9 NOTE 1 A procedure can only be renamed as a procedure. A function whose defining_designator is either an identifier or an operator_symbol can be renamed with either an identifier or an operator_symbol; for renaming as an operator, the subprogram specification given in the renaming_declaration is subject to the rules given in 6.6 for operator declarations. Enumeration literals can be renamed as functions; similarly, attribute_references that denote functions (such as references to Succ and Pred) can be renamed as functions. An entry can only be renamed as a procedure; the new name is only allowed to appear in contexts that allow a procedure name. An entry of a family can be renamed, but an entry family cannot be renamed as a whole.

10 NOTE 2 The operators of the root numeric types cannot be renamed because the types in the profile are anonymous, so the corresponding specifications cannot be written; the same holds for certain attributes, such as Pos.

11/5 *This paragraph was deleted.*

12 NOTE 3 The primitiveness of a renaming-as-declaration is determined by its profile, and by where it occurs, as for any declaration of (a view of) a subprogram; primitiveness is not determined by the renamed view. In order to perform a dispatching call, the subprogram name has to denote a primitive subprogram, not a nonprimitive renaming of a primitive subprogram.

Examples

13 *Examples of subprogram renaming declarations:*

14
```
procedure My_Write(C : in Character) renames Pool(K).Write; -- see 4.1.3
```

```
function Real_Plus(Left, Right : Real   ) return Real    renames "+";
function Int_Plus (Left, Right : Integer) return Integer renames "+";
```

```
function Rouge return Color renames Red;  -- see 3.5.1
function Rot   return Color renames Red;
function Rosso return Color renames Rouge;
```

```
function Next(X : Color) return Color renames Color'Succ; -- see 3.5.1
```

Example of a subprogram renaming declaration with new parameter names:

```
function "*" (X,Y : Vector) return Real renames Dot_Product; -- see 6.1
```

Example of a subprogram renaming declaration with a new default expression:

```
function Minimum(L : Link := Head) return Cell renames Min_Cell; -- see 6.1
```

8.5.5 Generic Renaming Declarations

A generic_renaming_declaration is used to rename a generic unit.

Syntax

generic_renaming_declaration ::=
  generic package defining_program_unit_name **renames** *generic_package*_name
    [aspect_specification];
 | **generic procedure** defining_program_unit_name **renames** *generic_procedure*_name
    [aspect_specification];
 | **generic function** defining_program_unit_name **renames** *generic_function*_name
    [aspect_specification];

Legality Rules

The renamed entity shall be a generic unit of the corresponding kind.

Static Semantics

A generic_renaming_declaration declares a new view of the renamed generic unit.

NOTE Although the properties of the new view are the same as those of the renamed view, the place where the generic_renaming_declaration occurs can affect the legality of subsequent renamings and instantiations that denote the generic_renaming_declaration, in particular if the renamed generic unit is a library unit (see 10.1.1).

Examples

Example of renaming a generic unit:

```
generic package Enum_IO renames Ada.Text_IO.Enumeration_IO;  -- see A.10.10
```

8.6 The Context of Overload Resolution

Because declarations can be overloaded, it is possible for an occurrence of a usage name to have more than one possible interpretation; in most cases, ambiguity is disallowed. This subclause describes how the possible interpretations resolve to the actual interpretation.

Certain rules of the language (the Name Resolution Rules) are considered “overloading rules”. If a possible interpretation violates an overloading rule, it is assumed not to be the intended interpretation; some other possible interpretation is assumed to be the actual interpretation. On the other hand, violations of nonoverloading rules do not affect which interpretation is chosen; instead, they cause the construct to be

illegal. To be legal, there usually has to be exactly one acceptable interpretation of a construct that is a "complete context", not counting any nested complete contexts.

3 The syntax rules of the language and the visibility rules given in 8.3 determine the possible interpretations. Most type checking rules (rules that require a particular type, or a particular class of types, for example) are overloading rules. Various rules for the matching of formal and actual parameters are overloading rules.

Name Resolution Rules

4 Overload resolution is applied separately to each *complete context*, not counting inner complete contexts. Each of the following constructs is a *complete context*:

5 • A context_item.

6 • A declarative_item or declaration.

7 • A statement.

8 • A pragma_argument_association.

9/4 • The *selecting*_expression of a case_statement or case_expression.

9.1/5 • The *variable*_name of an assignment_statement *A*, if the expression of *A* contains one or more target_names.

10 An (overall) *interpretation* of a complete context embodies its meaning, and includes the following information about the constituents of the complete context, not including constituents of inner complete contexts:

11 • for each constituent of the complete context, to which syntactic categories it belongs, and by which syntax rules; and

12 • for each usage name, which declaration it denotes (and, therefore, which view and which entity it denotes); and

13 • for a complete context that is a declarative_item, whether or not it is a completion of a declaration, and (if so) which declaration it completes.

14 A *possible interpretation* is one that obeys the syntax rules and the visibility rules. An *acceptable interpretation* is a possible interpretation that obeys the *overloading rules*, that is, those rules that specify an expected type or expected profile, or specify how a construct shall *resolve* or be *interpreted*.

15 The *interpretation* of a constituent of a complete context is determined from the overall interpretation of the complete context as a whole. Thus, for example, "interpreted as a function_call", means that the construct's interpretation says that it belongs to the syntactic category function_call.

16 Each occurrence of a usage name *denotes* the declaration determined by its interpretation. It also denotes the view declared by its denoted declaration, except in the following cases:

17/6 • If a usage name appears within the declarative region of a type_declaration and denotes that same type_declaration, then it denotes the *current instance* of the type (rather than the type itself); the current instance of a type is the object or value of the type that is associated with the execution that evaluates the usage name. Similarly, if a usage name appears within the declarative region of a subtype_declaration and denotes that same subtype_declaration, then it denotes the current instance of the subtype. These rules do not apply if the usage name appears within:

17.1/6 • the subtype_mark of an access_definition for an access-to-object type;

17.2/6 • the subtype_mark of a subtype_indication of an object_declaration;

- the subtype_mark of an object_renaming_declaration; or 17.3/6
- the subtype of a parameter or result of a callable entity or an access-to-subprogram type. 17.4/6

Within an aspect_specification for a type or subtype, the current instance represents a value of the type; it is not an object. Unless otherwise specified, the nominal subtype of this value is given by the subtype itself (the first subtype in the case of a type_declaration), prior to applying any predicate specified directly on the type or subtype. If the type or subtype is by-reference, the associated object of the value is the object associated (see 6.2) with the evaluation of the usage name. 17.5/6

- If a usage name appears within the declarative region of a single_task_declaration (or respectively, a single_protected_declaration) and it denotes that same single_task_declaration (respectively single_protected_declaration), then it denotes the *current instance* of the task unit (respectively protected unit), which in this case is the object declared by the unit. See also 9.1 and 9.5. 17.6/6
- If a usage name that is a direct_name appears within the declarative region of a type_declaration or single_protected_declaration and denotes a component of that same type_declaration or single_protected_declaration, then it is equivalent to a selected_component that denotes the corresponding component of the current instance of that type_declaration or of the object of that single_protected_declaration. This rule does not apply to a local_name within a record_representation_clause. 17.7/6
- If a usage name appears within the declarative region of a generic_declaration (but not within its generic_formal_part) and it denotes that same generic_declaration, then it denotes the *current instance* of the generic unit (rather than the generic unit itself). See also 12.3. 18

A usage name that denotes a view also denotes the entity of that view. 19

The *expected type* for a given expression, name, or other construct determines, according to the *type resolution rules* given below, the types considered for the construct during overload resolution. The type resolution rules provide support for class-wide programming, universal literals, dispatching operations, and anonymous access types: 20/2

- If a construct is expected to be of any type in a class of types, or of the universal or class-wide type for a class, then the type of the construct shall resolve to a type in that class or to a universal type that covers the class. 21
- If the expected type for a construct is a specific type *T*, then the type of the construct shall resolve either to *T*, or: 22
  - to *T*'Class; or 23
  - to a universal type that covers *T*; or 24
  - when *T* is a specific anonymous access-to-object type (see 3.10) with designated type *D*, to an access-to-object type whose designated type is *D*'Class or is covered by *D*; or 25/2
  - when *T* is a named general access-to-object type (see 3.10) with designated type *D*, to an anonymous access-to-object type whose designated type covers or is covered by *D*; or 25.1/3
  - when *T* is an anonymous access-to-subprogram type (see 3.10), to an access-to-subprogram type whose designated profile is type conformant with that of *T*. 25.2/3

In certain contexts, such as in a subprogram_renaming_declaration, the Name Resolution Rules define an *expected profile* for a given name; in such cases, the name shall resolve to the name of a callable entity whose profile is type conformant with the expected profile. 26

Legality Rules

27/6 When a construct is one that requires that its expected type be a *single* type in a given class, the type of the construct shall be determinable solely from the context in which the construct appears, excluding the construct itself, but using the requirement that it be in the given class. Furthermore, the context shall not be one that expects any type in some class that contains types of the given class; in particular, the construct shall not be the operand of a type_conversion. For the purposes of the application of subsequent rules, the construct is considered to have the single type so determined.

27.1/4 Other than for the *tested*_simple_expression of a membership test, if the expected type for a name or expression is not the same as the actual type of the name or expression, the actual type shall be convertible to the expected type (see 4.6); further, if the expected type is a named access-to-object type with designated type *D1* and the actual type is an anonymous access-to-object type with designated type *D2*, then *D1* shall cover *D2*, and the name or expression shall denote a view with an accessibility level for which the statically deeper relationship applies; in particular it shall not denote an access parameter nor a stand-alone access object.

28 A complete context shall have at least one acceptable interpretation; if there is exactly one, then that one is chosen.

29 There is a *preference* for the primitive operators (and ranges) of the root numeric types *root_integer* and *root_real*. In particular, if two acceptable interpretations of a constituent of a complete context differ only in that one is for a primitive operator (or range) of the type *root_integer* or *root_real*, and the other is not, the interpretation using the primitive operator (or range) of the root numeric type is *preferred*.

29.1/3 Similarly, there is a preference for the equality operators of the *universal_access* type (see 4.5.2). If two acceptable interpretations of a constituent of a complete context differ only in that one is for an equality operator of the *universal_access* type, and the other is not, the interpretation using the equality operator of the *universal_access* type is preferred.

30 For a complete context, if there is exactly one overall acceptable interpretation where each constituent's interpretation is the same as or preferred (in the above sense) over those in all other overall acceptable interpretations, then that one overall acceptable interpretation is chosen. Otherwise, the complete context is *ambiguous*.

31 A complete context other than a pragma_argument_association shall not be ambiguous.

32 A complete context that is a pragma_argument_association is allowed to be ambiguous (unless otherwise specified for the particular pragma), but only if every acceptable interpretation of the pragma argument is as a name that statically denotes a callable entity. Such a name denotes all of the declarations determined by its interpretations, and all of the views declared by these declarations.

33 NOTE If a usage name has only one acceptable interpretation, then it denotes the corresponding entity. However, this does not mean that the usage name is necessarily legal since other requirements exist which are not considered for overload resolution; for example, the fact that an expression is static, whether an object is constant, mode and subtype conformance rules, freezing rules, order of elaboration, and so on.

34 Similarly, subtypes are not considered for overload resolution (the violation of a constraint does not make a program illegal but raises an exception during program execution).

9 Tasks and Synchronization

The execution of an Ada program consists of the execution of one or more *tasks*. Each task represents a separable activity that proceeds independently and concurrently between the points where it *interacts* with other tasks. A single task, when within the context of a parallel construct, can represent multiple *logical threads of control* which can proceed in parallel; in other contexts, each task represents one logical thread of control. 1/5

The various forms of task interaction are described in this clause, and include: 1.1/5

- the activation and termination of a task; 2
- a call on a protected subprogram of a *protected object*, providing exclusive read-write access, or concurrent read-only access to shared data; 3
- a call on an entry, either of another task, allowing for synchronous communication with that task, or of a protected object, allowing for asynchronous communication with one or more other tasks using that same protected object; 4
- a timed operation, including a simple delay statement, a timed entry call or accept, or a timed asynchronous select statement (see next item); 5
- an asynchronous transfer of control as part of an asynchronous select statement, where a task stops what it is doing and begins execution at a different point in response to the completion of an entry call or the expiration of a delay; 6
- an abort statement, allowing one task to cause the termination of another task. 7

In addition, tasks can communicate indirectly by reading and updating (unprotected) shared variables, presuming the access is properly synchronized through some other kind of task interaction. 8

Static Semantics

The properties of a task are defined by a corresponding task declaration and task_body, which together define a program unit called a *task unit*. 9

Dynamic Semantics

Over time, tasks proceed through various *states*. A task is initially *inactive*; upon activation, and prior to its *termination* it is either *blocked* (as part of some task interaction) or *ready* to run. While ready, a task competes for the available *execution resources* that it requires to run. In the context of a parallel construct, a single task can utilize multiple processing resources simultaneously. 10/5

NOTE Concurrent task execution can be implemented on multicomputers, multiprocessors, or with interleaved execution on a single physical processor. On the other hand, whenever an implementation can determine that the required semantic effects can be achieved when parts of the execution of a single logical thread of control are performed by different physical processors acting in parallel, it can choose to perform them in this way. 11/5

9.1 Task Units and Task Objects

A task unit is declared by a *task declaration*, which has a corresponding task_body. A task declaration may be a task_type_declaration, in which case it declares a named task type; alternatively, it may be a single_task_declaration, in which case it defines an anonymous task type, as well as declaring a named task object of that type. 1

Syntax

2/3 task_type_declaration ::=
task type defining_identifier [known_discriminant_part]
[aspect_specification] [**is**
[**new** interface_list **with**]
task_definition];

3/5 single_task_declaration ::=
task defining_identifier
[aspect_specification] [**is**
[**new** interface_list **with**]
task_definition];

4 task_definition ::=
{task_item}
[**private**
{task_item}]
end [*task*_identifier]

5/1 task_item ::= entry_declaration | aspect_clause

6/3 task_body ::=
task body defining_identifier
[aspect_specification] **is**
declarative_part
begin
handled_sequence_of_statements
end [*task*_identifier];

7 If a *task*_identifier appears at the end of a task_definition or task_body, it shall repeat the defining_identifier.

Paragraph 8 was deleted.

Static Semantics

9 A task_definition defines a task type and its first subtype. The first list of task_items of a task_definition, together with the known_discriminant_part, if any, is called the visible part of the task unit. The optional list of task_items after the reserved word **private** is called the private part of the task unit.

9.1/1 For a task declaration without a task_definition, a task_definition without task_items is assumed.

9.2/3 For a task declaration with an interface_list, the task type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the task type or an access parameter designating the task type, and there is an entry_declaration for a single entry with the same identifier within the task declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming task entry using an implicitly declared nonabstract subprogram which has the same profile as the inherited subprogram and which overrides it.

Legality Rules

A task declaration requires a completion, which shall be a task_body, and every task_body shall be the completion of some task declaration. 9.3/2

Each *interface*_subtype_mark of an interface_list appearing within a task declaration shall denote a limited interface type that is not a protected interface. 9.4/2

The prefixed view profile of an explicitly declared primitive subprogram of a tagged task type shall not be type conformant with any entry of the task type, if the subprogram has the same defining name as the entry and the first parameter of the subprogram is of the task type or is an access parameter designating the task type. 9.5/3

For each primitive subprogram inherited by the type declared by a task declaration, at most one of the following shall apply: 9.6/2

- the inherited subprogram is overridden with a primitive subprogram of the task type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or 9.7/2
- the inherited subprogram is implemented by a single entry of the task type; in which case its prefixed view profile shall be subtype conformant with that of the task entry. 9.8/2

If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 9.9/2

Dynamic Semantics

The elaboration of a task declaration elaborates the task_definition. The elaboration of a single_task_declaration also creates an object of an (anonymous) task type. 10

The elaboration of a task_definition creates the task type and its first subtype; it also includes the elaboration of the entry_declarations in the given order. 11

As part of the initialization of a task object, any aspect_clauses and any per-object constraints associated with entry_declarations of the corresponding task_definition are elaborated in the given order. 12/1

The elaboration of a task_body has no effect other than to establish that tasks of the type can from then on be activated without failing the Elaboration_Check. 13

The execution of a task_body is invoked by the activation of a task of the corresponding type (see 9.2). 14

The content of a task object of a given task type includes: 15

- The values of the discriminants of the task object, if any; 16
- An entry queue for each entry of the task object; 17
- A representation of the state of the associated task. 18

NOTE 1 Other than in an access_definition, the name of a task unit within the declaration or body of the task unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding task type (and thus the name cannot be used as a subtype_mark). 19/2

NOTE 2 The notation of a selected_component can be used to denote a discriminant of a task (see 4.1.3). Within a task unit, the name of a discriminant of the task type denotes the corresponding discriminant of the current instance of the unit. 20

NOTE 3 A task type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators. If a programmer wants to write an application that stores and exchanges task identities, they can do so by defining an access type designating the corresponding task objects and by using access values for identification 21/5

purposes. Assignment is available for such an access type as for any access type. Alternatively, if the implementation supports the Systems Programming Annex, the Identity attribute can be used for task identification (see C.7.1).

Examples

22 *Examples of declarations of task types:*

23
```
task type Server is
   entry Next_Work_Item(WI : in Work_Item);
   entry Shut_Down;
end Server;
```

24/2
```
task type Keyboard_Driver(ID : Keyboard_ID := New_ID) is
      new Serial_Device with  -- see 3.9.4
   entry Read (C : out Character);
   entry Write(C : in  Character);
end Keyboard_Driver;
```

25 *Examples of declarations of single tasks:*

26
```
task Controller is
   entry Request(Level)(D : Item);  -- a family of entries
end Controller;
```

27
```
task Parser is
   entry Next_Lexeme(L : in  Lexical_Element);
   entry Next_Action(A : out Parser_Action);
end;
```

28
```
task User;  -- has no entries
```

29 *Examples of task objects:*

30
```
Agent    : Server;
Teletype : Keyboard_Driver(TTY_ID);
Pool     : array(1 .. 10) of Keyboard_Driver;
```

31 *Example of access type designating task objects:*

32
```
type Keyboard is access Keyboard_Driver;
Terminal : Keyboard := new Keyboard_Driver(Term_ID);
```

9.2 Task Execution - Task Activation

Dynamic Semantics

1 The execution of a task of a given task type consists of the execution of the corresponding task_body. The initial part of this execution is called the *activation* of the task; it consists of the elaboration of the declarative_part of the task_body. Should an exception be propagated by the elaboration of its declarative_part, the activation of the task is defined to have *failed*, and it becomes a completed task.

2/2 A task object (which represents one task) can be a part of a stand-alone object, of an object created by an allocator, or of an anonymous object of a limited type, or a coextension of one of these. All tasks that are part or coextensions of any of the stand-alone objects created by the elaboration of object_declarations (or generic_associations of formal objects of mode **in**) of a single declarative region are activated together. All tasks that are part or coextensions of a single object that is not a stand-alone object are activated together.

3/2 For the tasks of a given declarative region, the activations are initiated within the context of the handled_sequence_of_statements (and its associated exception_handlers if any — see 11.2), just prior to executing the statements of the handled_sequence_of_statements. For a package without an explicit body or an explicit handled_sequence_of_statements, an implicit body or an implicit null_statement is assumed, as defined in 7.2.

For tasks that are part or coextensions of a single object that is not a stand-alone object, activations are initiated after completing any initialization of the outermost object enclosing these tasks, prior to performing any other operation on the outermost object. In particular, for tasks that are part or coextensions of the object created by the evaluation of an allocator, the activations are initiated as the last step of evaluating the allocator, prior to returning the new access value. For tasks that are part or coextensions of an object that is the result of a function call, the activations are not initiated until after the function returns. 4/2

The task that created the new tasks and initiated their activations (the *activator*) is blocked until all of these activations complete (successfully or not). Once all of these activations are complete, if the activation of any of the tasks has failed (due to the propagation of an exception), Tasking_Error is raised in the activator, at the place at which it initiated the activations. Otherwise, the activator proceeds with its execution normally. Any tasks that are aborted prior to completing their activation are ignored when determining whether to raise Tasking_Error. 5

If the master that directly encloses the point where the activation of a task *T* would be initiated, completes before the activation of *T* is initiated, *T* becomes terminated and is never activated. Furthermore, if a return statement is left such that the return object is not returned to the caller, any task that was created as a part of the return object or one of its coextensions immediately becomes terminated and is never activated. 6/3

NOTE 1 An entry of a task can be called before the task has been activated. 7

NOTE 2 If several tasks are activated together, the execution of any of these tasks can proceed without waiting until the end of the activation of the other tasks. 8/5

NOTE 3 A task can become completed during its activation either because of an exception or because it is aborted (see 9.8). 9

Examples

Example of task activation: 10

```
procedure P is                                                    11
   A, B : Server;      -- elaborate the task objects A, B
   C    : Server;      -- elaborate the task object C
begin
   -- the tasks A, B, C are activated together before the first statement
   ...
end;
```

9.3 Task Dependence - Termination of Tasks

Dynamic Semantics

Each task (other than an environment task — see 10.2) *depends* on one or more masters (see 7.6.1), as follows: 1

- If the task is created by the evaluation of an allocator for a given named access type, it depends on each master that includes the elaboration of the declaration of the ultimate ancestor of the given access type. 2/4
- If the task is created by the elaboration of an object_declaration, it depends on each master that includes this elaboration. 3
- Otherwise, the task depends on the master of the outermost object of which it is a part (as determined by the accessibility level of that object — see 3.10.2 and 7.6.1), as well as on any master whose execution includes that of the master of the outermost object. 3.1/2

4 Furthermore, if a task depends on a given master, it is defined to depend on the task that executes the master, and (recursively) on any master of that task.

5 A task is said to be *completed* when the execution of its corresponding task_body is completed. A task is said to be *terminated* when any finalization of the task_body has been performed (see 7.6.1). The first step of finalizing a master (including a task_body) is to wait for the termination of any tasks dependent on the master. The task executing the master is blocked until all the dependents have terminated. Any remaining finalization is then performed and the master is left.

6/1 Completion of a task (and the corresponding task_body) can occur when the task is blocked at a select_statement with an open terminate_alternative (see 9.7.1); the open terminate_alternative is selected if and only if the following conditions are satisfied:

7/2 • The task depends on some completed master; and

8 • Each task that depends on the master considered is either already terminated or similarly blocked at a select_statement with an open terminate_alternative.

9 When both conditions are satisfied, the task considered becomes completed, together with all tasks that depend on the master considered that are not yet completed.

10 NOTE 1 The full view of a limited private type can be a task type, or can have subcomponents of a task type. Creation of an object of such a type creates dependences according to the full type.

11 NOTE 2 An object_renaming_declaration defines a new view of an existing entity and hence creates no further dependence.

12/5 NOTE 3 The rules given for the collective completion of a group of tasks all blocked on select_statements with open terminate_alternatives ensure that the collective completion can occur only when there are no remaining active tasks that can call one of the tasks being collectively completed.

13 NOTE 4 If two or more tasks are blocked on select_statements with open terminate_alternatives, and become completed collectively, their finalization actions proceed concurrently.

14 NOTE 5 The completion of a task can occur due to any of the following:

15 • the raising of an exception during the elaboration of the declarative_part of the corresponding task_body;

16 • the completion of the handled_sequence_of_statements of the corresponding task_body;

17 • the selection of an open terminate_alternative of a select_statement in the corresponding task_body;

18 • the abort of the task.

Examples

19 *Example of task dependence:*

20
```
declare
   type Global is access Server;         -- see 9.1
   A, B : Server;
   G    : Global;
begin
   -- activation of A and B
   declare
      type Local is access Server;
      X : Global := new Server;  -- activation of X.all
      L : Local  := new Server;  -- activation of L.all
      C : Server;
   begin
      -- activation of C
      G := X;  -- both G and X designate the same task object
      ...
   end;  -- await termination of C and L.all (but not X.all)
   ...
end;  -- await termination of A, B, and G.all
```

9.4 Protected Units and Protected Objects

A *protected object* provides coordinated access to shared data, through calls on its visible *protected operations*, which can be *protected subprograms* or *protected entries*. A *protected unit* is declared by a *protected declaration*, which has a corresponding protected_body. A protected declaration may be a protected_type_declaration, in which case it declares a named protected type; alternatively, it may be a single_protected_declaration, in which case it defines an anonymous protected type, as well as declaring a named protected object of that type. 1

Syntax

protected_type_declaration ::= 2/3
 protected type defining_identifier [known_discriminant_part]
 [aspect_specification] **is**
 [**new** interface_list **with**]
 protected_definition;

single_protected_declaration ::= 3/3
 protected defining_identifier
 [aspect_specification] **is**
 [**new** interface_list **with**]
 protected_definition;

protected_definition ::= 4
 { protected_operation_declaration }
[**private**
 { protected_element_declaration }]
 end [*protected*_identifier]

protected_operation_declaration ::= subprogram_declaration 5/1
 | entry_declaration
 | aspect_clause

protected_element_declaration ::= protected_operation_declaration 6
 | component_declaration

protected_body ::= 7/3
 protected body defining_identifier
 [aspect_specification] **is**
 { protected_operation_item }
 end [*protected*_identifier];

protected_operation_item ::= subprogram_declaration 8/4
 | subprogram_body
 | null_procedure_declaration
 | expression_function_declaration
 | entry_body
 | aspect_clause

If a *protected*_identifier appears at the end of a protected_definition or protected_body, it shall repeat the defining_identifier. 9

Paragraph 10 was deleted.

Static Semantics

11/2 A protected_definition defines a protected type and its first subtype. The list of protected_operation_declarations of a protected_definition, together with the known_discriminant_part, if any, is called the visible part of the protected unit. The optional list of protected_element_declarations after the reserved word **private** is called the private part of the protected unit.

11.1/3 For a protected declaration with an interface_list, the protected type inherits user-defined primitive subprograms from each progenitor type (see 3.9.4), in the same way that a derived type inherits user-defined primitive subprograms from its progenitor types (see 3.4). If the first parameter of a primitive inherited subprogram is of the protected type or an access parameter designating the protected type, and there is a protected_operation_declaration for a protected subprogram or single entry with the same identifier within the protected declaration, whose profile is type conformant with the prefixed view profile of the inherited subprogram, the inherited subprogram is said to be *implemented* by the conforming protected subprogram or entry using an implicitly declared nonabstract subprogram which has the same profile as the inherited subprogram and which overrides it.

Legality Rules

11.2/2 A protected declaration requires a completion, which shall be a protected_body, and every protected_body shall be the completion of some protected declaration.

11.3/2 Each *interface*_subtype_mark of an interface_list appearing within a protected declaration shall denote a limited interface type that is not a task interface.

11.4/3 The prefixed view profile of an explicitly declared primitive subprogram of a tagged protected type shall not be type conformant with any protected operation of the protected type, if the subprogram has the same defining name as the protected operation and the first parameter of the subprogram is of the protected type or is an access parameter designating the protected type.

11.5/2 For each primitive subprogram inherited by the type declared by a protected declaration, at most one of the following shall apply:

11.6/2 - the inherited subprogram is overridden with a primitive subprogram of the protected type, in which case the overriding subprogram shall be subtype conformant with the inherited subprogram and not abstract; or

11.7/2 - the inherited subprogram is implemented by a protected subprogram or single entry of the protected type, in which case its prefixed view profile shall be subtype conformant with that of the protected subprogram or entry.

11.8/2 If neither applies, the inherited subprogram shall be a null procedure. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

11.9/3 If an inherited subprogram is implemented by a protected procedure or an entry, then the first parameter of the inherited subprogram shall be of mode **out** or **in out**, or an access-to-variable parameter. If an inherited subprogram is implemented by a protected function, then the first parameter of the inherited subprogram shall be of mode **in**, but not an access-to-variable parameter.

11.10/2 If a protected subprogram declaration has an overriding_indicator, then at the point of the declaration:

11.11/2 - if the overriding_indicator is **overriding**, then the subprogram shall implement an inherited subprogram;

- if the overriding_indicator is **not overriding**, then the subprogram shall not implement any inherited subprogram. 11.12/2

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 11.13/2

Dynamic Semantics

The elaboration of a protected declaration elaborates the protected_definition. The elaboration of a single_protected_declaration also creates an object of an (anonymous) protected type. 12

The elaboration of a protected_definition creates the protected type and its first subtype; it also includes the elaboration of the component_declarations and protected_operation_declarations in the given order. 13

As part of the initialization of a protected object, any per-object constraints (see 3.8) are elaborated. 14

The elaboration of a protected_body has no other effect than to establish that protected operations of the type can from then on be called without failing the Elaboration_Check. 15

The content of an object of a given protected type includes: 16

- The values of the components of the protected object, including (implicitly) an entry queue for each entry declared for the protected object; 17
- A representation of the state of the execution resource *associated* with the protected object (one such resource is associated with each protected object). 18

The execution resource associated with a protected object has to be acquired to read or update any components of the protected object; it can be acquired (as part of a protected action — see 9.5.1) either for concurrent read-only access, or for exclusive read-write access. 19

As the first step of the *finalization* of a protected object, each call remaining on any entry queue of the object is removed from its queue and Program_Error is raised at the place of the corresponding entry_call_statement. 20

Bounded (Run-Time) Errors

It is a bounded error to call an entry or subprogram of a protected object after that object is finalized. If the error is detected, Program_Error is raised. Otherwise, the call proceeds normally, which may leave a task queued forever. 20.1/2

NOTE 1 Within the declaration or body of a protected unit other than in an access_definition, the name of the protected unit denotes the current instance of the unit (see 8.6), rather than the first subtype of the corresponding protected type (and thus the name cannot be used as a subtype_mark). 21/2

NOTE 2 A selected_component can be used to denote a discriminant of a protected object (see 4.1.3). Within a protected unit, the name of a discriminant of the protected type denotes the corresponding discriminant of the current instance of the unit. 22

NOTE 3 A protected type is a limited type (see 7.5), and hence precludes use of assignment_statements and predefined equality operators. 23/2

NOTE 4 The bodies of the protected operations given in the protected_body define the actions that take place upon calls to the protected operations. 24

NOTE 5 The declarations in the private part are only visible within the private part and the body of the protected unit. 25

Examples

26 *Example of declaration of protected type and corresponding body:*

27
```
protected type Resource is
   entry Seize;
   procedure Release;
private
   Busy : Boolean := False;
end Resource;
```

28
```
protected body Resource is
   entry Seize when not Busy is
   begin
      Busy := True;
   end Seize;
```

29
```
   procedure Release is
   begin
      Busy := False;
   end Release;
end Resource;
```

30 *Example of a single protected declaration and corresponding body:*

31/5
```
protected Shared_Array is
   --  Index, Item, and Item_Array are global types
   function  Component    (N : in Index) return Item;
   procedure Set_Component(N : in Index; E : in  Item);
private
   Table : Item_Array(Index) := (others => Null_Item);
end Shared_Array;
```

32
```
protected body Shared_Array is
   function Component(N : in Index) return Item is
   begin
      return Table(N);
   end Component;
```

33
```
   procedure Set_Component(N : in Index; E : in Item) is
   begin
      Table(N) := E;
   end Set_Component;
end Shared_Array;
```

34 *Examples of protected objects:*

35
```
Control  : Resource;
Flags    : array(1 .. 100) of Resource;
```

9.5 Intertask Communication

1 The primary means for intertask communication is provided by calls on entries and protected subprograms. Calls on protected subprograms allow coordinated access to shared data objects. Entry calls allow for blocking the caller until a given condition is satisfied (namely, that the corresponding entry is open — see 9.5.3), and then communicating data or control information directly with another task or indirectly via a shared protected object.

Static Semantics

2/3 When a name or prefix denotes an entry, protected subprogram, or a prefixed view of a primitive subprogram of a limited interface whose first parameter is a controlling parameter, the name or prefix determines a *target object*, as follows:

3/3 • If it is a direct_name or expanded name that denotes the declaration (or body) of the operation, then the target object is implicitly specified to be the current instance of the task or protected

unit immediately enclosing the operation; a call using such a name is defined to be an *internal call*;

- If it is a selected_component that is not an expanded name, then the target object is explicitly specified to be the object denoted by the prefix of the name; a call using such a name is defined to be an *external call*; 4/3
- If the name or prefix is a dereference (implicit or explicit) of an access-to-protected-subprogram value, then the target object is determined by the prefix of the Access attribute_reference that produced the access value originally; a call using such a name is defined to be an *external call*; 5/3
- If the name or prefix denotes a subprogram_renaming_declaration, then the target object is as determined by the name of the renamed entity. 6

A call on an entry or a protected subprogram either uses a name or prefix that determines a target object implicitly, as above, or is a call on (a non-prefixed view of) a primitive subprogram of a limited interface whose first parameter is a controlling parameter, in which case the target object is identified explicitly by the first parameter. This latter case is an *external call*. 6.1/3

A corresponding definition of target object applies to a requeue_statement (see 9.5.4), with a corresponding distinction between an *internal requeue* and an *external requeue*. 7

Legality Rules

If a name or prefix determines a target object, and the name denotes a protected entry or procedure, then the target object shall be a variable, unless the prefix is for an attribute_reference to the Count attribute (see 9.9). 7.1/3

An internal call on a protected function shall not occur within: 7.2/6

- a precondition expression (see 6.1.1) of a protected operation; 7.3/6
- the expression of a Dynamic_Predicate aspect (see 3.2.4) of a protected type; 7.4/6
- the expression of a Default_Initial_Condition aspect (see 7.3.3) of a protected type; nor 7.5/6
- a default_expression of a parameter_specification of a protected operation. 7.6/6

Dynamic Semantics

Within the body of a protected operation, the current instance (see 8.6) of the immediately enclosing protected unit is determined by the target object specified (implicitly or explicitly) in the call (or requeue) on the protected operation. 8

Any call on a protected procedure or entry of a target protected object is defined to be an update to the object, as is a requeue on such an entry. 9

Syntax

synchronization_kind ::= By_Entry | By_Protected_Procedure | Optional 10/3

Static Semantics

For the declaration of a primitive procedure of a synchronized tagged type the following language-defined representation aspect may be specified with an aspect_specification (see 13.1.1): 11/3

Synchronization 12/3

If specified, the aspect definition shall be a synchronization_kind.

13/3 Inherited subprograms inherit the Synchronization aspect, if any, from the corresponding subprogram of the parent or progenitor type. If an overriding operation does not have a directly specified Synchronization aspect then the Synchronization aspect of the inherited operation is inherited by the overriding operation.

Legality Rules

14/3 The synchronization_kind By_Protected_Procedure shall not be applied to a primitive procedure of a task interface.

15/3 A procedure for which the specified synchronization_kind is By_Entry shall be implemented by an entry. A procedure for which the specified synchronization_kind is By_Protected_Procedure shall be implemented by a protected procedure. A procedure for which the specified synchronization_kind is Optional may be implemented by an entry or by a procedure (including a protected procedure).

16/3 If a primitive procedure overrides an inherited operation for which the Synchronization aspect has been specified to be By_Entry or By_Protected_Procedure, then any specification of the aspect Synchronization applied to the overriding operation shall have the same synchronization_kind.

17/3 In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Static Semantics

18/5 For a program unit, task entry, formal package, formal subprogram, formal object of an anonymous access-to-subprogram type, enumeration literal, and for a subtype (including a formal subtype), the following language-defined operational aspect is defined:

19/5 Nonblocking This aspect specifies the blocking restriction for the entity; it shall be specified by a static Boolean expression. The aspect_definition can be omitted from the specification of this aspect; in that case, the aspect for the entity is True.

20/5 The Nonblocking aspect may be specified for all entities for which it is defined, except for protected operations and task entries. In particular, Nonblocking may be specified for generic formal parameters.

21/5 When aspect Nonblocking is False for an entity, the entity can contain a potentially blocking operation; such an entity *allows blocking*. If the aspect is True for an entity, the entity is said to be *nonblocking*.

22/5 For a generic instantiation and entities declared within such an instance, the aspect is determined by the Nonblocking aspect for the corresponding entity of the generic unit, **and**ed with the Nonblocking aspects of the actual generic parameters *used* by the entity. If the aspect is directly specified for an instance, the specified expression shall have the same value as the Nonblocking aspect of the instance (after **and**ing with the aspects of the used actual parameters). In the absence of a Use_Formal aspect, all actual generic parameters are presumed to be *used* by an entity (see H.7.1).

23/5 For a (protected or task) entry, the Nonblocking aspect is False.

24/5 For an enumeration literal, the Nonblocking aspect is True.

25/5 For a predefined operator of an elementary type, the Nonblocking aspect is True. For a predefined operator of a composite type, the Nonblocking aspect of the operator is the same as the Nonblocking aspect for the type.

26/5 For a dereference of an access-to-subprogram type, the Nonblocking aspect of the designated subprogram is that of the access-to-subprogram type.

27/5 For the base subtype of a scalar (sub)type, the Nonblocking aspect is True.

For an inherited primitive dispatching subprogram that is null or abstract, the subprogram is nonblocking if and only if a corresponding subprogram of at least one ancestor is nonblocking. For any other inherited subprogram, it is nonblocking if and only if the corresponding subprogram of the parent is nonblocking. 28/5

Unless directly specified, overridings of dispatching operations inherit this aspect. 29/5

Unless directly specified, for a formal subtype, formal package, or formal subprogram, the Nonblocking aspect is that of the actual subtype, package, or subprogram. 30/5

Unless directly specified, for a nonfirst subtype *S*, the Nonblocking aspect is that of the subtype identified in the subtype_indication defining *S*; unless directly specified for the first subtype of a derived type, the Nonblocking aspect is that of the ancestor subtype. 31/5

Unless directly specified, for any other program unit, first subtype, or formal object, the Nonblocking aspect of the entity is determined by the Nonblocking aspect for the innermost program unit enclosing the entity. 32/5

If not specified for a library unit, the Nonblocking aspect is True if the library unit is declared pure, or False otherwise. 33/5

The following are defined to be *potentially blocking* operations: 34/5

- a select_statement; 35/5
- an accept_statement; 36/5
- an entry_call_statement, or a call on a procedure that renames or is implemented by an entry; 37/5
- a delay_statement; 38/5
- an abort_statement; 39/5
- task creation or activation; 40/5
- during a protected action, an external call on a protected subprogram (or an external requeue) with the same target object as that of the protected action. 41/5

If a language-defined subprogram allows blocking, then a call on the subprogram is a potentially blocking operation. 42/5

Legality Rules

A portion of program text is called a *nonblocking region* if it is anywhere within a parallel construct, or if the innermost enclosing program unit is nonblocking. A nonblocking region shall not contain any of the following: 43/5

- a select_statement; 44/5
- an accept_statement; 45/5
- a delay_statement; 46/5
- an abort_statement; 47/5
- task creation or activation. 48/5

Furthermore, a parallel construct shall neither contain a call on a callable entity for which the Nonblocking aspect is False, nor shall it contain a call on a callable entity declared within a generic unit that uses a generic formal parameter with Nonblocking aspect False (see Use_Formal aspect in H.7.1). 49/5

Finally, a nonblocking region that is outside of a parallel construct shall not contain a call on a callable entity for which the Nonblocking aspect is False, unless the region is within a generic unit and the callable 50/5

entity is associated with a generic formal parameter of the generic unit, or the call is within the aspect_definition of an assertion aspect for an entity that allows blocking.

51/5 For the purposes of the above rules, an entry_body is considered nonblocking if the immediately enclosing protected unit is nonblocking.

52/5 For a subtype for which aspect Nonblocking is True, any predicate expression that applies to the subtype shall only contain constructs that are allowed immediately within a nonblocking program unit.

53/5 A subprogram shall be nonblocking if it overrides a nonblocking dispatching operation. An entry shall not implement a nonblocking procedure. If an inherited dispatching subprogram allows blocking, then the corresponding subprogram of each ancestor shall allow blocking.

54/5 It is illegal to directly specify aspect Nonblocking for the first subtype of the full view of a type that has a partial view. If the Nonblocking aspect of the full view is inherited, it shall have the same value as that of the partial view, or have the value True.

55/5 Aspect Nonblocking shall be directly specified for the first subtype of a derived type only if it has the same value as the Nonblocking aspect of the ancestor subtype or if it is specified True. Aspect Nonblocking shall be directly specified for a nonfirst subtype *S* only if it has the same value as the Nonblocking aspect of the subtype identified in the subtype_indication defining *S* or if it is specified True.

56/5 For an access-to-object type that is nonblocking, the Allocate, Deallocate, and Storage_Size operations on its storage pool shall be nonblocking.

57/5 For a composite type that is nonblocking:

58/5 - All component subtypes shall be nonblocking;

59/5 - For a record type or extension, every call in the default_expression of a component (including discriminants) shall call an operation that is nonblocking;

60/5 - For a controlled type, the Initialize, Finalize, and Adjust (if any) subprograms shall be nonblocking.

61/5 The predefined equality operator for a composite type, unless it is for a record type or record extension and the operator is overridden by a primitive equality operator, is illegal if it is nonblocking and:

62/5 - for a record type or record extension, the parent primitive "=" allows blocking; or

63/5 - some component is of a type *T*, and:

64/5 - *T* is a record type or record extension that has a primitive "=" that allows blocking; or

65/5 - *T* is neither a record type nor a record extension, and *T* has a predefined "=" that allows blocking.

66/5 In a generic instantiation:

67/5 - the actual subprogram corresponding to a nonblocking formal subprogram shall be nonblocking (an actual that is an entry is not permitted in this case);

68/5 - the actual subtype corresponding to a nonblocking formal subtype shall be nonblocking;

69/5 - the actual object corresponding to a formal object of a nonblocking access-to-subprogram type shall be of a nonblocking access-to-subprogram type;

70/5 - the actual instance corresponding to a nonblocking formal package shall be nonblocking.

71/5 In addition to the places where Legality Rules normally apply (see 12.3), the above rules also apply in the private part of an instance of a generic unit.

NOTE The synchronization_kind By_Protected_Procedure implies that the operation will not block. 72/3

9.5.1 Protected Subprograms and Protected Actions

A *protected subprogram* is a subprogram declared immediately within a protected_definition. Protected procedures provide exclusive read-write access to the data of a protected object; protected functions provide concurrent read-only access to the data. 1

Static Semantics

Within the body of a protected function (or a function declared immediately within a protected_body), the current instance of the enclosing protected unit is defined to be a constant (that is, its subcomponents may be read but not updated). Within the body of a protected procedure (or a procedure declared immediately within a protected_body), and within an entry_body, the current instance is defined to be a variable (updating is permitted). 2

For a type declared by a protected_type_declaration or for the anonymous type of an object declared by a single_protected_declaration, the following language-defined type-related representation aspect may be specified: 2.1/4

Exclusive_Functions 2.2/4

The type of aspect Exclusive_Functions is Boolean. If not specified (including by inheritance), the aspect is False.

A value of True for this aspect indicates that protected functions behave in the same way as protected procedures with respect to mutual exclusion and queue servicing (see below). 2.3/4

A protected procedure or entry is an *exclusive* protected operation. A protected function of a protected type *P* is an exclusive protected operation if the Exclusive_Functions aspect of *P* is True. 2.4/4

Dynamic Semantics

For the execution of a call on a protected subprogram, the evaluation of the name or prefix and of the parameter associations, and any assigning back of **in out** or **out** parameters, proceeds as for a normal subprogram call (see 6.4). If the call is an internal call (see 9.5), the body of the subprogram is executed as for a normal subprogram call. If the call is an external call, then the body of the subprogram is executed as part of a new *protected action* on the target protected object; the protected action completes after the body of the subprogram is executed. A protected action can also be started by an entry call (see 9.5.3). 3

A new protected action is not started on a protected object while another protected action on the same protected object is underway, unless both actions are the result of a call on a nonexclusive protected function. This rule is expressible in terms of the execution resource associated with the protected object: 4/4

- *Starting* a protected action on a protected object corresponds to *acquiring* the execution resource associated with the protected object, either for exclusive read-write access if the protected action is for a call on an exclusive protected operation, or for concurrent read-only access otherwise; 5/4
- *Completing* the protected action corresponds to *releasing* the associated execution resource. 6

After performing an exclusive protected operation on a protected object, but prior to completing the associated protected action, the entry queues (if any) of the protected object are serviced (see 9.5.3). 7/4

If a parallel construct occurs within a protected action, no new logical threads of control are created. Instead, each element of the parallel construct that would have become a separate logical thread of control executes on the logical thread of control that is performing the protected action. If there are multiple such elements initiated at the same point, they execute in an arbitrary order. 7.1/5

Bounded (Run-Time) Errors

8/5 During a protected action, it is a bounded error to invoke an operation that is potentially blocking (see 9.5).

Paragraphs 9 through 16 were moved to 9.5.

17/5 If the bounded error is detected, Program_Error is raised. If not detected, the bounded error can result in deadlock or a (nested) protected action on the same target object.

18/5 During a protected action, a call on a subprogram whose body contains a potentially blocking operation is a bounded error. If the bounded error is detected, Program_Error is raised; otherwise, the call proceeds normally.

19/5 NOTE 1 If two tasks both try to start a protected action on a protected object, and at most one is calling a protected nonexclusive function, then only one of the tasks can proceed. Although the other task cannot proceed, it is not considered blocked, and it can be consuming processing resources while it awaits its turn. Unless there is an admission policy (see D.4.1) in effect, there is no language-defined ordering or queuing presumed for tasks competing to start a protected action — on a multiprocessor such tasks can use busy-waiting; for further monoprocessor and multiprocessor considerations, see D.3, “Priority Ceiling Locking”.

20/5 NOTE 2 The body of a protected unit can contain declarations and bodies for local subprograms. These are not visible outside the protected unit.

21 NOTE 3 The body of a protected function can contain internal calls on other protected functions, but not protected procedures, because the current instance is a constant. On the other hand, the body of a protected procedure can contain internal calls on both protected functions and procedures.

22 NOTE 4 From within a protected action, an internal call on a protected subprogram, or an external call on a protected subprogram with a different target object is not considered a potentially blocking operation.

23/5 NOTE 5 The aspect Nonblocking can be specified True on the definition of a protected unit in order to reject most attempts to use potentially blocking operations within the protected unit (see 9.5). The pragma Detect_Blocking can be used to ensure that any remaining executions of potentially blocking operations during a protected action raise Program_Error. See H.5.

Examples

24 *Examples of protected subprogram calls (see 9.4):*

25
```
Shared_Array.Set_Component(N, E);
E := Shared_Array.Component(M);
Control.Release;
```

9.5.2 Entries and Accept Statements

1 Entry_declarations, with the corresponding entry_bodies or accept_statements, are used to define potentially queued operations on tasks and protected objects.

Syntax

2/3
```
entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile
    [aspect_specification];
```

3
```
accept_statement ::=
  accept entry_direct_name [(entry_index)] parameter_profile [do
   handled_sequence_of_statements
  end [entry_identifier]];
```

entry_index ::= expression 4

entry_body ::= 5/5
  entry defining_identifier entry_body_formal_part
    [aspect_specification]
  entry_barrier **is**
    declarative_part
  begin
    handled_sequence_of_statements
  end [*entry_*identifier];

entry_body_formal_part ::= [(entry_index_specification)] parameter_profile 6

entry_barrier ::= **when** condition 7

entry_index_specification ::= 8/5
  for defining_identifier **in** discrete_subtype_definition [aspect_specification]

If an *entry_*identifier appears at the end of an accept_statement, it shall repeat the *entry_*direct_name. If an *entry_*identifier appears at the end of an entry_body, it shall repeat the defining_identifier. 9

An entry_declaration is allowed only in a protected or task declaration. 10

An overriding_indicator is not allowed in an entry_declaration that includes a discrete_subtype_definition. 10.1/2

Name Resolution Rules

In an accept_statement, the expected profile for the *entry_*direct_name is that of the entry_declaration; the expected type for an entry_index is that of the subtype defined by the discrete_subtype_definition of the corresponding entry_declaration. 11

Within the handled_sequence_of_statements of an accept_statement, if a selected_component has a prefix that denotes the corresponding entry_declaration, then the entity denoted by the prefix is the accept_statement, and the selected_component is interpreted as an expanded name (see 4.1.3); the selector_name of the selected_component has to be the identifier for some formal parameter of the accept_statement. 12

Legality Rules

An entry_declaration in a task declaration shall not contain a specification for an access parameter (see 3.10). 13

If an entry_declaration has an overriding_indicator, then at the point of the declaration: 13.1/2

- if the overriding_indicator is **overriding**, then the entry shall implement an inherited subprogram; 13.2/2
- if the overriding_indicator is **not overriding**, then the entry shall not implement any inherited subprogram. 13.3/2

In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 13.4/2

For an accept_statement, the innermost enclosing body shall be a task_body, and the *entry_*direct_name shall denote an entry_declaration in the corresponding task declaration; the profile of the accept_- 14

statement shall conform fully to that of the corresponding entry_declaration. An accept_statement shall have a parenthesized entry_index if and only if the corresponding entry_declaration has a discrete_subtype_definition.

15 An accept_statement shall not be within another accept_statement that corresponds to the same entry_declaration, nor within an asynchronous_select inner to the enclosing task_body.

16 An entry_declaration of a protected unit requires a completion, which shall be an entry_body, and every entry_body shall be the completion of an entry_declaration of a protected unit. The profile of the entry_body shall conform fully to that of the corresponding declaration.

17 An entry_body_formal_part shall have an entry_index_specification if and only if the corresponding entry_declaration has a discrete_subtype_definition. In this case, the discrete_subtype_definitions of the entry_declaration and the entry_index_specification shall fully conform to one another (see 6.3.1).

18 A name that denotes a formal parameter of an entry_body is not allowed within the entry_barrier of the entry_body.

Static Semantics

19 The parameter modes defined for parameters in the parameter_profile of an entry_declaration are the same as for a subprogram_declaration and have the same meaning (see 6.2).

20 An entry_declaration with a discrete_subtype_definition (see 3.6) declares a *family* of distinct entries having the same profile, with one such entry for each value of the *entry index subtype* defined by the discrete_subtype_definition. A name for an entry of a family takes the form of an indexed_component, where the prefix denotes the entry_declaration for the family, and the index value identifies the entry within the family. The term *single entry* is used to refer to any entry other than an entry of an entry family.

21 In the entry_body for an entry family, the entry_index_specification declares a named constant whose subtype is the entry index subtype defined by the corresponding entry_declaration; the value of the *named entry index* identifies which entry of the family was called.

Dynamic Semantics

22/1 The elaboration of an entry_declaration for an entry family consists of the elaboration of the discrete_subtype_definition, as described in 3.8. The elaboration of an entry_declaration for a single entry has no effect.

23 The actions to be performed when an entry is called are specified by the corresponding accept_statements (if any) for an entry of a task unit, and by the corresponding entry_body for an entry of a protected unit.

23.1/5 The interaction between a task that calls an entry and an accepting task is called a *rendezvous*.

24/5 For the execution of an accept_statement, the entry_index, if any, is first evaluated and converted to the entry index subtype; this index value identifies which entry of the family is to be accepted. Further execution of the accept_statement is then blocked until a caller of the corresponding entry is selected (see 9.5.3), whereupon the handled_sequence_of_statements, if any, of the accept_statement is executed, with the formal parameters associated with the corresponding actual parameters of the selected entry call. Execution of the rendezvous consists of the execution of the handled_sequence_of_statements, performance of any postcondition or type invariant checks associated with the entry, and any initialization or finalization associated with these checks, as described in 6.1.1 and 7.3.2. After execution of the rendezvous, the accept_statement completes and is left. The two tasks then proceed independently. When

an exception is propagated from the handled_sequence_of_statements of an accept_statement, the same exception is also raised by the execution of the corresponding entry_call_statement.

This paragraph was deleted. 25/5

An entry_body is executed when the condition of the entry_barrier evaluates to True and a caller of the corresponding single entry, or entry of the corresponding entry family, has been selected (see 9.5.3). For the execution of the entry_body, the declarative_part of the entry_body is elaborated, and the handled_sequence_of_statements of the body is executed, as for the execution of a subprogram_body. The value of the named entry index, if any, is determined by the value of the entry index specified in the *entry*_name of the selected entry call (or intermediate requeue_statement — see 9.5.4). 26

NOTE 1 A task entry has corresponding accept_statements (zero or more), whereas a protected entry has a corresponding entry_body (exactly one). 27

NOTE 2 A consequence of the rule regarding the allowed placements of accept_statements is that a task can execute accept_statements only for its own entries. 28

NOTE 3 A return statement (see 6.5) or a requeue_statement (see 9.5.4) can be used to complete the execution of an accept_statement or an entry_body. 29/5

NOTE 4 The condition in the entry_barrier can reference anything visible except the formal parameters of the entry. This includes the entry index (if any), the components (including discriminants) of the protected object, the Count attribute of an entry of that protected object, and data global to the protected unit. 30/5

The restriction against referencing the formal parameters within an entry_barrier ensures that all calls of the same entry see the same barrier value. If it is necessary to look at the parameters of an entry call before deciding whether to handle it, the entry_barrier can be "**when** True" and the caller can be requeued (on some private entry) when its parameters indicate that it cannot be handled immediately. 31

Examples

Examples of entry declarations: 32

```
entry Read(V : out Item);                          33
entry Seize;
entry Request(Level)(D : Item);   -- a family of entries
```

Examples of accept statements: 34

```
accept Shut_Down;                                  35

accept Read(V : out Item) do                       36
   V := Local_Item;
end Read;

accept Request(Low)(D : Item) do                   37
   ...
end Request;
```

9.5.3 Entry Calls

An entry_call_statement (an *entry call*) can appear in various contexts. A *simple* entry call is a stand-alone statement that represents an unconditional call on an entry of a target task or a protected object. Entry calls can also appear as part of select_statements (see 9.7). 1

Syntax

entry_call_statement ::= *entry*_name [actual_parameter_part]; 2

Name Resolution Rules

The *entry*_name given in an entry_call_statement shall resolve to denote an entry. The rules for parameter associations are the same as for subprogram calls (see 6.4 and 6.4.1). 3

Static Semantics

4 The *entry_*name of an entry_call_statement specifies (explicitly or implicitly) the target object of the call, the entry or entry family, and the entry index, if any (see 9.5).

Dynamic Semantics

5 Under certain circumstances (detailed below), an entry of a task or protected object is checked to see whether it is *open* or *closed*:

6/3 • An entry of a task is open if the task is blocked on an accept_statement that corresponds to the entry (see 9.5.2), or on a selective_accept (see 9.7.1) with an open accept_alternative that corresponds to the entry; otherwise, it is closed.

7/3 • An entry of a protected object is open if the condition of the entry_barrier of the corresponding entry_body evaluates to True; otherwise, it is closed. If the evaluation of the condition propagates an exception, the exception Program_Error is propagated to all current callers of all entries of the protected object.

8 For the execution of an entry_call_statement, evaluation of the name and of the parameter associations is as for a subprogram call (see 6.4). The entry call is then *issued*: For a call on an entry of a protected object, a new protected action is started on the object (see 9.5.1). The named entry is checked to see if it is open; if open, the entry call is said to be *selected immediately*, and the execution of the call proceeds as follows:

9 • For a call on an open entry of a task, the accepting task becomes ready and continues the execution of the corresponding accept_statement (see 9.5.2).

10 • For a call on an open entry of a protected object, the corresponding entry_body is executed (see 9.5.2) as part of the protected action.

11 If the accept_statement or entry_body completes other than by a requeue (see 9.5.4), return is made to the caller (after servicing the entry queues — see below); any necessary assigning back of formal to actual parameters occurs, as for a subprogram call (see 6.4.1); such assignments take place outside of any protected action.

12 If the named entry is closed, the entry call is added to an *entry queue* (as part of the protected action, for a call on a protected entry), and the call remains queued until it is selected or cancelled; there is a separate (logical) entry queue for each entry of a given task or protected object (including each entry of an entry family).

13 When a queued call is *selected*, it is removed from its entry queue. Selecting a queued call from a particular entry queue is called *servicing* the entry queue. An entry with queued calls can be serviced under the following circumstances:

14 • When the associated task reaches a corresponding accept_statement, or a selective_accept with a corresponding open accept_alternative;

15/4 • If after performing, as part of a protected action on the associated protected object, an exclusive protected operation on the object, the entry is checked and found to be open.

16 If there is at least one call on a queue corresponding to an open entry, then one such call is selected according to the *entry queuing policy* in effect (see below), and the corresponding accept_statement or entry_body is executed as above for an entry call that is selected immediately.

17 The entry queuing policy controls selection among queued calls both for task and protected entry queues. The default entry queuing policy is to select calls on a given entry queue in order of arrival. If calls from two or more queues are simultaneously eligible for selection, the default entry queuing policy does not specify which queue is serviced first. Other entry queuing policies can be specified by pragmas (see D.4).

For a protected object, the above servicing of entry queues continues until there are no open entries with queued calls, at which point the protected action completes. 18

For an entry call that is added to a queue, and that is not the triggering_statement of an asynchronous_select (see 9.7.4), the calling task is blocked until the call is cancelled, or the call is selected and a corresponding accept_statement or entry_body completes without requeuing. In addition, the calling task is blocked during a rendezvous. 19

An attempt can be made to cancel an entry call upon an abort (see 9.8) and as part of certain forms of select_statement (see 9.7.2, 9.7.3, and 9.7.4). The cancellation does not take place until a point (if any) when the call is on some entry queue, and not protected from cancellation as part of a requeue (see 9.5.4); at such a point, the call is removed from the entry queue and the call completes due to the cancellation. The cancellation of a call on an entry of a protected object is a protected action, and as such cannot take place while any other protected action is occurring on the protected object. Like any protected action, it includes servicing of the entry queues (in case some entry barrier depends on a Count attribute). 20

A call on an entry of a task that has already completed its execution raises the exception Tasking_Error at the point of the call; similarly, this exception is raised at the point of the call if the called task completes its execution or becomes abnormal before accepting the call or completing the rendezvous (see 9.8). This applies equally to a simple entry call and to an entry call as part of a select_statement. 21

Implementation Permissions

An implementation may perform the sequence of steps of a protected action using any thread of control; it can be a thread other than that of the task that started the protected action. If an entry_body completes without requeuing, then the corresponding calling task may be made ready without waiting for the entire protected action to complete. 22/5

When the entry of a protected object is checked to see whether it is open, the implementation can bypass reevaluating the condition of the corresponding entry_barrier if no variable or attribute referenced by the condition (directly or indirectly) has been altered by the execution (or cancellation) of a call to an exclusive protected operation of the object since the condition was last evaluated. 23/5

An implementation may evaluate the conditions of all entry_barriers of a given protected object any time any entry of the object is checked to see if it is open. 24

When an attempt is made to cancel an entry call, the implementation can use a thread of control other than that of the task (or interrupt) that initiated the cancellation; in particular, it may use the thread of control of the caller itself to attempt the cancellation, even if this can allow the entry call to be selected in the interim. 25/5

NOTE 1 If an exception is raised during the execution of an entry_body, it is propagated to the corresponding caller (see 11.4). 26

NOTE 2 For a call on a protected entry, the entry is checked to see if it is open prior to queuing the call, and again thereafter if its Count attribute (see 9.9) is referenced in some entry barrier. 27

NOTE 3 In addition to simple entry calls, the language permits timed, conditional, and asynchronous entry calls (see 9.7.2, 9.7.3, and see 9.7.4). 28

NOTE 4 The condition of an entry_barrier is allowed to be evaluated by an implementation more often than strictly necessary, even if the evaluation can have side effects. On the other hand, an implementation can avoid reevaluating the condition if nothing it references was updated by an intervening protected action on the protected object, even if the condition references some global variable that is updated by an action performed from outside of a protected action. 29/5

Examples

30 *Examples of entry calls:*

31
```
Agent.Shut_Down;                        -- see 9.1
Parser.Next_Lexeme(E);                  -- see 9.1
Pool(5).Read(Next_Char);                -- see 9.1
Controller.Request(Low)(Some_Item);     -- see 9.1
Flags(3).Seize;                         -- see 9.4
```

9.5.4 Requeue Statements

1 A requeue_statement can be used to complete an accept_statement or entry_body, while redirecting the corresponding entry call to a new (or the same) entry queue. Such a *requeue* can be performed with or without allowing an intermediate cancellation of the call, due to an abort or the expiration of a delay.

Syntax

2/3 requeue_statement ::= **requeue** *procedure_or_entry_*name [**with abort**];

Name Resolution Rules

3/3 The *procedure_or_entry_*name of a requeue_statement shall resolve to denote a procedure or an entry (the *requeue target*). The profile of the entry, or the profile or prefixed profile of the procedure, shall either have no parameters, or be type conformant (see 6.3.1) with the profile of the innermost enclosing entry_body or accept_statement.

Legality Rules

4 A requeue_statement shall be within a callable construct that is either an entry_body or an accept_statement, and this construct shall be the innermost enclosing body or callable construct.

5/3 If the requeue target has parameters, then its (prefixed) profile shall be subtype conformant with the profile of the innermost enclosing callable construct.

5.1/4 Given a requeue_statement where the innermost enclosing callable construct is for an entry *E1*, for every specific or class-wide postcondition expression *P1* that applies to *E1*, there shall exist a postcondition expression *P2* that applies to the requeue target *E2* such that

5.2/4 - *P1* is fully conformant with the expression produced by replacing each reference in *P2* to a formal parameter of *E2* with a reference to the corresponding formal paramter of *E1*; and

5.3/4 - if *P1* is enabled, then *P2* is also enabled.

5.4/5 The requeue target shall not have an applicable specific or class-wide postcondition that includes an Old or Index attribute_reference.

5.5/4 If the requeue target is declared immediately within the task_definition of a named task type or the protected_definition of a named protected type, and if the requeue statement occurs within the body of that type, and if the requeue is an external requeue, then the requeue target shall not have a specific or class-wide postcondition which includes a name denoting either the current instance of that type or any entity declared within the declaration of that type.

5.6/4 If the target is a procedure, the name shall denote a renaming of an entry, or shall denote a view or a prefixed view of a primitive subprogram of a synchronized interface, where the first parameter of the unprefixed view of the primitive subprogram shall be a controlling parameter, and the Synchronization aspect shall be specified with synchronization_kind By_Entry for the primitive subprogram.

In a requeue_statement of an accept_statement of some task unit, either the target object shall be a part of a formal parameter of the accept_statement, or the accessibility level of the target object shall not be equal to or statically deeper than any enclosing accept_statement of the task unit. In a requeue_-statement of an entry_body of some protected unit, either the target object shall be a part of a formal parameter of the entry_body, or the accessibility level of the target object shall not be statically deeper than that of the entry_declaration for the entry_body. 6/3

Dynamic Semantics

The execution of a requeue_statement begins with the following sequence of steps: 7/5

1. The *procedure_or_entry*_name is evaluated. This includes evaluation of the prefix (if any) identifying the target task or protected object and of the expression (if any) identifying the entry within an entry family. 7.1/5
2. If the target object is not a part of a formal parameter of the innermost enclosing callable construct, a check is made that the accessibility level of the target object is not equal to or deeper than the level of the innermost enclosing callable construct. If this check fails, Program_Error is raised. 7.2/5
3. Precondition checks are performed as for a call to the requeue target. 7.3/5
4. The entry_body or accept_statement enclosing the requeue_statement is then completed, finalized, and left (see 7.6.1). 7.4/5

For the execution of a requeue on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the requeued call is either selected immediately or queued, as for a normal entry call (see 9.5.3). 8

For the execution of a requeue on an entry of a target protected object, after leaving the enclosing callable construct: 9

- if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object — see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1); 10
- if the requeue is an external requeue (that is, the target protected object is not implicitly the same as the current object — see 9.5), a protected action is started on the target object and proceeds as for a normal entry call (see 9.5.3). 11

If the requeue target named in the requeue_statement has formal parameters, then during the execution of the accept_statement or entry_body corresponding to the new entry and during the checking of any preconditions of the new entry, the formal parameters denote the same objects as did the corresponding formal parameters of the callable construct completed by the requeue. In any case, no parameters are specified in a requeue_statement; any parameter passing is implicit. 12/4

If the requeue_statement includes the reserved words **with abort** (it is a *requeue-with-abort*), then: 13

- if the original entry call has been aborted (see 9.8), then the requeue acts as an abort completion point for the call, and the call is cancelled and no requeue is performed; 14
- if the original entry call was timed (or conditional), then the original expiration time is the expiration time for the requeued call. 15

If the reserved words **with abort** do not appear, then the call remains protected against cancellation while queued as the result of the requeue_statement. 16

NOTE A requeue is permitted from a single entry to an entry of an entry family, or vice versa. The entry index, if any, plays no part in the subtype conformance check between the profiles of the two entries; an entry index is part of the *entry*_name for an entry of a family. 17/5

Examples

18 *Examples of requeue statements:*

19
```
requeue Request(Medium) with abort;
                    -- requeue on a member of an entry family of the current task, see 9.1
```

20
```
requeue Flags(I).Seize;
                    -- requeue on an entry of an array component, see 9.4
```

9.6 Delay Statements, Duration, and Time

1 A delay_statement is used to block further execution until a specified *expiration time* is reached. The expiration time can be specified either as a particular point in time (in a delay_until_statement), or in seconds from the current time (in a delay_relative_statement). The language-defined package Calendar provides definitions for a type Time and associated operations, including a function Clock that returns the current time.

Syntax

2 delay_statement ::= delay_until_statement | delay_relative_statement

3 delay_until_statement ::= **delay until** *delay*_expression;

4 delay_relative_statement ::= **delay** *delay*_expression;

Name Resolution Rules

5 The expected type for the *delay*_expression in a delay_relative_statement is the predefined type Duration. The *delay*_expression in a delay_until_statement is expected to be of any nonlimited type.

Legality Rules

6/3 There can be multiple time bases, each with a corresponding clock, and a corresponding *time type*. The type of the *delay*_expression in a delay_until_statement shall be a time type — either the type Time defined in the language-defined package Calendar (see below), the type Time in the package Real_Time (see D.8), or some other implementation-defined time type.

Static Semantics

7 There is a predefined fixed point type named Duration, declared in the visible part of package Standard; a value of type Duration is used to represent the length of an interval of time, expressed in seconds. The type Duration is not specific to a particular time base, but can be used with any time base.

8/3 A value of the type Time in package Calendar, or of some other time type, represents a time as reported by a corresponding clock.

9 The following language-defined library package exists:

10/5
```
package Ada.Calendar
  with Nonblocking, Global => in out synchronized is
  type Time is private;

11/2
  subtype Year_Number  is Integer range 1901 .. 2399;
  subtype Month_Number is Integer range 1 .. 12;
  subtype Day_Number   is Integer range 1 .. 31;
  subtype Day_Duration is Duration range 0.0 .. 86_400.0;

12
  function Clock return Time;
```

```
   function Year   (Date : Time) return Year_Number;                                    13
   function Month  (Date : Time) return Month_Number;
   function Day    (Date : Time) return Day_Number;
   function Seconds(Date : Time) return Day_Duration;

   procedure Split (Date  : in Time;                                                    14
                    Year    : out Year_Number;
                    Month   : out Month_Number;
                    Day     : out Day_Number;
                    Seconds : out Day_Duration);

   function Time_Of(Year  : Year_Number;                                                15
                    Month   : Month_Number;
                    Day     : Day_Number;
                    Seconds : Day_Duration := 0.0)
     return Time;

   function "+" (Left : Time;   Right : Duration) return Time;                          16
   function "+" (Left : Duration; Right : Time) return Time;
   function "-" (Left : Time;   Right : Duration) return Time;
   function "-" (Left : Time;   Right : Time) return Duration;

   function "<" (Left, Right : Time) return Boolean;                                    17
   function "<="(Left, Right : Time) return Boolean;
   function ">" (Left, Right : Time) return Boolean;
   function ">="(Left, Right : Time) return Boolean;

   Time_Error : exception;                                                              18

private                                                                                 19
   ... -- not specified by the language
end Ada.Calendar;
```

Dynamic Semantics

For the execution of a delay_statement, the *delay*_expression is first evaluated. For a delay_until_statement, the expiration time for the delay is the value of the *delay*_expression, in the time base associated with the type of the expression. For a delay_relative_statement, the expiration time is defined as the current time, in the time base associated with relative delays, plus the value of the *delay*_expression converted to the type Duration, and then rounded up to the next clock tick. The time base associated with relative delays is as defined in D.9, “Delay Accuracy” or is implementation defined. 20

The task executing a delay_statement is blocked until the expiration time is reached, at which point it becomes ready again. If the expiration time has already passed, the task is not blocked. 21

If an attempt is made to *cancel* the delay_statement (as part of an asynchronous_select or abort — see 9.7.4 and 9.8), the statement is cancelled if the expiration time has not yet passed, thereby completing the delay_statement. 22/3

The time base associated with the type Time of package Calendar is implementation defined. The function Clock of package Calendar returns a value representing the current time for this time base. The implementation-defined value of the named number System.Tick (see 13.7) is an approximation of the length of the real-time interval during which the value of Calendar.Clock remains constant. 23

The functions Year, Month, Day, and Seconds return the corresponding values for a given value of the type Time, as appropriate to an implementation-defined time zone; the procedure Split returns all four corresponding values. Conversely, the function Time_Of combines a year number, a month number, a day number, and a duration, into a value of type Time. The operators "+" and "–" for addition and subtraction of times and durations, and the relational operators for times, have the conventional meaning. 24/2

If Time_Of is called with a seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a seconds value of 0.0. The value returned by the function Seconds or through the Seconds parameter of the procedure Split is always less than 86_400.0. 25

26/1 The exception Time_Error is raised by the function Time_Of if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "–" if the result is not representable in the type Time or Duration, as appropriate. This exception is also raised by the functions Year, Month, Day, and Seconds and the procedure Split if the year number of the given date is outside of the range of the subtype Year_Number.

Implementation Requirements

27 The implementation of the type Duration shall allow representation of time intervals (both positive and negative) up to at least 86400 seconds (one day); Duration'Small shall not be greater than twenty milliseconds. The implementation of the type Time shall allow representation of all dates with year numbers in the range of Year_Number; it may allow representation of other dates as well (both earlier and later).

Implementation Permissions

28/3 An implementation may define additional time types.

29 An implementation may raise Time_Error if the value of a *delay_*expression in a delay_until_statement of a select_statement represents a time more than 90 days past the current time. The actual limit, if any, is implementation-defined.

Implementation Advice

30 Whenever possible in an implementation, the value of Duration'Small should be no greater than 100 microseconds.

31/5 The time base for delay_relative_statements should be monotonic; it can be different than the time base as used for Calendar.Clock.

32 NOTE 1 A delay_relative_statement with a negative value of the *delay_*expression is equivalent to one with a zero value.

33/5 NOTE 2 A delay_statement can be executed by the environment task; consequently delay_statements can be executed as part of the elaboration of a library_item or the execution of the main subprogram. Such statements delay the environment task (see 10.2).

34 NOTE 3 A delay_statement is an abort completion point and a potentially blocking operation, even if the task is not actually blocked.

35 NOTE 4 There is no necessary relationship between System.Tick (the resolution of the clock of package Calendar) and Duration'Small (the *small* of type Duration).

36 NOTE 5 Additional requirements associated with delay_statements are given in D.9, “Delay Accuracy”.

Examples

37 *Example of a relative delay statement:*

38
```
delay 3.0;     -- delay 3.0 seconds
```

Example of a periodic task:

```
declare
   use Ada.Calendar;
   Next_Time : Time := Clock + Period;
                       -- Period is a global constant of type Duration
begin
   loop                -- repeated every Period seconds
      delay until Next_Time;
      ... -- perform some actions
      Next_Time := Next_Time + Period;
   end loop;
end;
```

9.6.1 Formatting, Time Zones, and other operations for Time

Static Semantics

The following language-defined library packages exist:

```
package Ada.Calendar.Time_Zones
   with Nonblocking, Global => in out synchronized is

   -- Time zone manipulation:

   type Time_Offset is range -28*60 .. 28*60;

   Unknown_Zone_Error : exception;

   function Local_Time_Offset (Date : Time := Clock) return Time_Offset;

   function UTC_Time_Offset (Date : Time := Clock) return Time_Offset
      renames Local_Time_Offset;

end Ada.Calendar.Time_Zones;


package Ada.Calendar.Arithmetic
   with Nonblocking, Global => in out synchronized is

   -- Arithmetic on days:

   type Day_Count is range
     -366*(1+Year_Number'Last - Year_Number'First)
     ..
     366*(1+Year_Number'Last - Year_Number'First);

   subtype Leap_Seconds_Count is Integer range -2047 .. 2047;

   procedure Difference (Left, Right : in Time;
                         Days : out Day_Count;
                         Seconds : out Duration;
                         Leap_Seconds : out Leap_Seconds_Count);

   function "+" (Left : Time; Right : Day_Count) return Time;
   function "+" (Left : Day_Count; Right : Time) return Time;
   function "-" (Left : Time; Right : Day_Count) return Time;
   function "-" (Left, Right : Time) return Day_Count;

end Ada.Calendar.Arithmetic;


with Ada.Calendar.Time_Zones;
package Ada.Calendar.Formatting
   with Nonblocking, Global => in out synchronized is

   -- Day of the week:

   type Day_Name is (Monday, Tuesday, Wednesday, Thursday,
       Friday, Saturday, Sunday);

   function Day_Of_Week (Date : Time) return Day_Name;

   function Day_Of_Week (Date : Time;
      Time_Zone : Time_Zones.Time_Offset) return Day_Name;
```

19/2
```
   -- Hours:Minutes:Seconds access:
```
20/2
```
   subtype Hour_Number         is Natural range 0 .. 23;
   subtype Minute_Number       is Natural range 0 .. 59;
   subtype Second_Number       is Natural range 0 .. 59;
   subtype Second_Duration     is Day_Duration range 0.0 .. 1.0;
```
21/2
```
   function Year       (Date : Time;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Year_Number;
```
22/2
```
   function Month      (Date : Time;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Month_Number;
```
23/2
```
   function Day        (Date : Time;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Day_Number;
```
23.1/6
```
   function Seconds_Of (Date : Time;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Day_Duration;
```
24/2
```
   function Hour       (Date : Time;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Hour_Number;
```
25/2
```
   function Minute     (Date : Time;
                        Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Minute_Number;
```
26/2
```
   function Second     (Date : Time)
                           return Second_Number;
```
27/2
```
   function Sub_Second (Date : Time)
                           return Second_Duration;
```
28/2
```
   function Seconds_Of (Hour   :  Hour_Number;
                        Minute : Minute_Number;
                        Second : Second_Number := 0;
                        Sub_Second : Second_Duration := 0.0)
                           return Day_Duration;
```
29/2
```
   procedure Split (Seconds    : in Day_Duration;
                    Hour       : out Hour_Number;
                    Minute     : out Minute_Number;
                    Second     : out Second_Number;
                    Sub_Second : out Second_Duration);
```
30/2
```
   function Time_Of (Year       : Year_Number;
                     Month      : Month_Number;
                     Day        : Day_Number;
                     Hour       : Hour_Number;
                     Minute     : Minute_Number;
                     Second     : Second_Number;
                     Sub_Second : Second_Duration := 0.0;
                     Leap_Second: Boolean := False;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Time;
```
31/2
```
   function Time_Of (Year       : Year_Number;
                     Month      : Month_Number;
                     Day        : Day_Number;
                     Seconds    : Day_Duration := 0.0;
                     Leap_Second: Boolean := False;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                           return Time;
```

```
   procedure Split (Date       : in Time;                                    32/2
                    Year       : out Year_Number;
                    Month      : out Month_Number;
                    Day        : out Day_Number;
                    Hour       : out Hour_Number;
                    Minute     : out Minute_Number;
                    Second     : out Second_Number;
                    Sub_Second : out Second_Duration;
                    Time_Zone  : in Time_Zones.Time_Offset := 0);
   procedure Split (Date       : in Time;                                    33/2
                    Year       : out Year_Number;
                    Month      : out Month_Number;
                    Day        : out Day_Number;
                    Hour       : out Hour_Number;
                    Minute     : out Minute_Number;
                    Second     : out Second_Number;
                    Sub_Second : out Second_Duration;
                    Leap_Second: out Boolean;
                    Time_Zone  : in Time_Zones.Time_Offset := 0);
   procedure Split (Date       : in Time;                                    34/2
                    Year       : out Year_Number;
                    Month      : out Month_Number;
                    Day        : out Day_Number;
                    Seconds    : out Day_Duration;
                    Leap_Second: out Boolean;
                    Time_Zone  : in Time_Zones.Time_Offset := 0);
   -- Simple image and value:                                              35/2
   function Image (Date : Time;
                   Include_Time_Fraction : Boolean := False;
                   Time_Zone  : Time_Zones.Time_Offset := 0) return String;
   function Local_Image (Date : Time;                                       35.1/5
                         Include_Time_Fraction : Boolean := False)
      return String is
      (Image (Date, Include_Time_Fraction,
              Time_Zones.Local_Time_Offset (Date)));
   function Value (Date : String;                                           36/2
                   Time_Zone  : Time_Zones.Time_Offset := 0) return Time;
   function Image (Elapsed_Time : Duration;                                 37/2
                   Include_Time_Fraction : Boolean := False) return String;
   function Value (Elapsed_Time : String) return Duration;                  38/2
end Ada.Calendar.Formatting;                                                39/2
```

Type Time_Offset represents for a given locality at a given moment the number of minutes the local time is, at that moment, ahead (+) or behind (-) Coordinated Universal Time (abbreviated UTC). The Time_Offset for UTC is zero. 40/5

```
function Local_Time_Offset (Date : Time := Clock) return Time_Offset;       41/5
```

Returns, as a number of minutes, the Time_Offset of the implementation-defined time zone of Calendar, at the time Date. If the time zone of the Calendar implementation is unknown, then Unknown_Zone_Error is raised. 42/5

```
procedure Difference (Left, Right : in Time;                                43/2
                      Days : out Day_Count;
                      Seconds : out Duration;
                      Leap_Seconds : out Leap_Seconds_Count);
```

Returns the difference between Left and Right. Days is the number of days of difference, Seconds is the remainder seconds of difference excluding leap seconds, and Leap_Seconds is the number of leap seconds. If Left < Right, then Seconds <= 0.0, Days <= 0, and Leap_Seconds <= 0. Otherwise, all values are nonnegative. The absolute value of Seconds is always less than 44/2

86_400.0. For the returned values, if Days = 0, then Seconds + Duration(Leap_Seconds) = Calendar."–" (Left, Right).

45/2
```
function "+" (Left : Time; Right : Day_Count) return Time;
function "+" (Left : Day_Count; Right : Time) return Time;
```

46/2 Adds a number of days to a time value. Time_Error is raised if the result is not representable as a value of type Time.

47/2
```
function "-" (Left : Time; Right : Day_Count) return Time;
```

48/2 Subtracts a number of days from a time value. Time_Error is raised if the result is not representable as a value of type Time.

49/2
```
function "-" (Left, Right : Time) return Day_Count;
```

50/2 Subtracts two time values, and returns the number of days between them. This is the same value that Difference would return in Days.

51/2
```
function Day_Of_Week (Date : Time) return Day_Name;
```

52/2 Returns the day of the week for Time. This is based on the Year, Month, and Day values of Time.

52.1/6
```
function Day_Of_Week (Date : Time;
   Time_Zone : Time_Zones.Time_Offset) return Day_Name;
```

52.2/6 Returns the day of the week for Time in the given Time_Zone.

53/2
```
function Year       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Year_Number;
```

54/2 Returns the year for Date, as appropriate for the specified time zone offset.

55/2
```
function Month      (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Month_Number;
```

56/2 Returns the month for Date, as appropriate for the specified time zone offset.

57/2
```
function Day        (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Day_Number;
```

58/2 Returns the day number for Date, as appropriate for the specified time zone offset.

58.1/6
```
function Seconds_Of (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Day_Duration;
```

58.2/6 Returns the number of seconds in Date, as appropriate for the specified time zone offset.

59/2
```
function Hour       (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Hour_Number;
```

60/2 Returns the hour for Date, as appropriate for the specified time zone offset.

61/2
```
function Minute     (Date : Time;
                     Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Minute_Number;
```

62/2 Returns the minute within the hour for Date, as appropriate for the specified time zone offset.

```
function Second     (Date : Time)                                    63/2
                       return Second_Number;
```

Returns the second within the hour and minute for Date. 64/2

```
function Sub_Second (Date : Time)                                    65/2
                       return Second_Duration;
```

Returns the fraction of second for Date (this has the same accuracy as Day_Duration). The value returned is always less than 1.0. 66/2

```
function Seconds_Of (Hour   : Hour_Number;                           67/2
                     Minute : Minute_Number;
                     Second : Second_Number := 0;
                     Sub_Second : Second_Duration := 0.0)
                       return Day_Duration;
```

Returns a Day_Duration value for the combination of the given Hour, Minute, Second, and Sub_Second. This value can be used in Calendar.Time_Of as well as the argument to Calendar."+" and Calendar."–". If Seconds_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Seconds_Of for the next second with a Sub_Second value of 0.0. 68/2

```
procedure Split (Seconds    : in Day_Duration;                       69/2
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration);
```

Splits Seconds into Hour, Minute, Second and Sub_Second in such a way that the resulting values all belong to their respective subtypes. The value returned in the Sub_Second parameter is always less than 1.0. If Seconds = 86400.0, Split propagates Time_Error. 70/3

```
function Time_Of (Year       : Year_Number;                          71/2
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Hour       : Hour_Number;
                  Minute     : Minute_Number;
                  Second     : Second_Number;
                  Sub_Second : Second_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Time;
```

If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Sub_Second value of 1.0, the value returned is equal to the value of Time_Of for the next second with a Sub_Second value of 0.0. 72/2

```
function Time_Of (Year       : Year_Number;                          73/2
                  Month      : Month_Number;
                  Day        : Day_Number;
                  Seconds    : Day_Duration := 0.0;
                  Leap_Second: Boolean := False;
                  Time_Zone  : Time_Zones.Time_Offset := 0)
                        return Time;
```

If Leap_Second is False, returns a Time built from the date and time values, relative to the specified time zone offset. If Leap_Second is True, returns the Time that represents the time within the leap second that is one second later than the time specified by the other parameters. 74/2

Time_Error is raised if the parameters do not form a proper date or time. If Time_Of is called with a Seconds value of 86_400.0, the value returned is equal to the value of Time_Of for the next day with a Seconds value of 0.0.

75/2
```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

76/2 If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Sub_Second parameter is always less than 1.0.

77/2
```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Hour       : out Hour_Number;
                 Minute     : out Minute_Number;
                 Second     : out Second_Number;
                 Sub_Second : out Second_Duration;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

78/2 Splits Date into its constituent parts (Year, Month, Day, Hour, Minute, Second, Sub_Second), relative to the specified time zone offset. The value returned in the Sub_Second parameter is always less than 1.0.

79/2
```
procedure Split (Date       : in Time;
                 Year       : out Year_Number;
                 Month      : out Month_Number;
                 Day        : out Day_Number;
                 Seconds    : out Day_Duration;
                 Leap_Second: out Boolean;
                 Time_Zone  : in Time_Zones.Time_Offset := 0);
```

80/2 If Date does not represent a time within a leap second, splits Date into its constituent parts (Year, Month, Day, Seconds), relative to the specified time zone offset, and sets Leap_Second to False. If Date represents a time within a leap second, set the constituent parts to values corresponding to a time one second earlier than that given by Date, relative to the specified time zone offset, and sets Leap_Seconds to True. The value returned in the Seconds parameter is always less than 86_400.0.

81/2
```
function Image (Date : Time;
                Include_Time_Fraction : Boolean := False;
                Time_Zone  : Time_Zones.Time_Offset := 0) return String;
```

82/2 Returns a string form of the Date relative to the given Time_Zone. The format is "Year-Month-Day Hour:Minute:Second", where the Year is a 4-digit value, and all others are 2-digit values, of the functions defined in Calendar and Calendar.Formatting, including a leading zero, if needed. The separators between the values are a minus, another minus, a colon, and a single space

between the Day and Hour. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value.

```
function Value (Date : String;                                                    83/2
                Time_Zone  : Time_Zones.Time_Offset := 0) return Time;
```

Returns a Time value for the image given as Date, relative to the given time zone. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Time value. 84/2

```
function Image (Elapsed_Time : Duration;                                          85/2
                Include_Time_Fraction : Boolean := False) return String;
```

Returns a string form of the Elapsed_Time. The format is "Hour:Minute:Second", where all values are 2-digit values, including a leading zero, if necessary. The separators between the values are colons. If Include_Time_Fraction is True, the integer part of Sub_Seconds*100 is suffixed to the string as a point followed by a 2-digit value. If Elapsed_Time < 0.0, the result is Image (**abs** Elapsed_Time, Include_Time_Fraction) prefixed with a minus sign. If **abs** Elapsed_Time represents 100 hours or more, the result is implementation-defined. 86/5

```
function Value (Elapsed_Time : String) return Duration;                           87/2
```

Returns a Duration value for the image given as Elapsed_Time. Constraint_Error is raised if the string is not formatted as described for Image, or the function cannot interpret the given string as a Duration value. 88/2

Implementation Advice

An implementation should support leap seconds if the target system supports them. If leap seconds are not supported, Difference should return zero for Leap_Seconds, Split should return False for Leap_Second, and Time_Of should raise Time_Error if Leap_Second is True. 89/2

NOTE 1 The implementation-defined time zone of package Calendar can be the local time zone. Local_Time_Offset always returns the difference relative to the implementation-defined time zone of package Calendar. If Local_Time_Offset does not raise Unknown_Zone_Error, UTC time can be safely calculated (within the accuracy of the underlying time-base). 90/5

NOTE 2 Calling Split on the results of subtracting Duration(Local_Time_Offset*60) from Clock provides the components (hours, minutes, and so on) of the UTC time. In the United States, for example, Local_Time_Offset will generally be negative. 91/5

9.7 Select Statements

There are four forms of the select_statement. One form provides a selective wait for one or more select_alternatives. Two provide timed and conditional entry calls. The fourth provides asynchronous transfer of control. 1

Syntax

select_statement ::= 2
  selective_accept
 | timed_entry_call
 | conditional_entry_call
 | asynchronous_select

Examples

Example of a select statement:

```
select
   accept Driver_Awake_Signal;
or
   delay 30.0*Seconds;
   Stop_The_Train;
end select;
```

9.7.1 Selective Accept

This form of the select_statement allows a combination of waiting for, and selecting from, one or more alternatives. The selection may depend on conditions associated with each alternative of the selective_accept.

Syntax

selective_accept ::=
 select
  [guard]
   select_alternative
{ **or**
  [guard]
   select_alternative }
[**else**
  sequence_of_statements]
 end select;

guard ::= **when** condition =>

select_alternative ::=
  accept_alternative
 | delay_alternative
 | terminate_alternative

accept_alternative ::=
 accept_statement [sequence_of_statements]

delay_alternative ::=
 delay_statement [sequence_of_statements]

terminate_alternative ::= **terminate**;

A selective_accept shall contain at least one accept_alternative. In addition, it can contain:

- a terminate_alternative (only one); or
- one or more delay_alternatives; or
- an *else part* (the reserved word **else** followed by a sequence_of_statements).

These three possibilities are mutually exclusive.

Legality Rules

If a selective_accept contains more than one delay_alternative, then all shall be delay_relative_statements, or all shall be delay_until_statements for the same time type.

Dynamic Semantics

A select_alternative is said to be *open* if it is not immediately preceded by a guard, or if the condition of its guard evaluates to True. It is said to be *closed* otherwise. 14

For the execution of a selective_accept, any guard conditions are evaluated; open alternatives are thus determined. For an open delay_alternative, the *delay*_expression is also evaluated. Similarly, for an open accept_alternative for an entry of a family, the entry_index is also evaluated. These evaluations are performed in an arbitrary order, except that a *delay*_expression or entry_index is not evaluated until after evaluating the corresponding condition, if any. Selection and execution of one open alternative, or of the else part, then completes the execution of the selective_accept; the rules for this selection are described below. 15

Open accept_alternatives are first considered. Selection of one such alternative takes place immediately if the corresponding entry already has queued calls. If several alternatives can thus be selected, one of them is selected according to the entry queuing policy in effect (see 9.5.3 and D.4). When such an alternative is selected, the selected call is removed from its entry queue and the handled_sequence_of_statements (if any) of the corresponding accept_statement is executed; after the rendezvous completes any subsequent sequence_of_statements of the alternative is executed. If no selection is immediately possible (in the above sense) and there is no else part, the task blocks until an open alternative can be selected. 16

Selection of the other forms of alternative or of an else part is performed as follows: 17

- An open delay_alternative is selected when its expiration time is reached if no accept_alternative or other delay_alternative can be selected prior to the expiration time. If several delay_alternatives have this same expiration time, one of them is selected according to the queuing policy in effect (see D.4); the default queuing policy chooses arbitrarily among the delay_alternatives whose expiration time has passed. 18
- The else part is selected and its sequence_of_statements is executed if no accept_alternative can immediately be selected; in particular, if all alternatives are closed. 19
- An open terminate_alternative is selected if the conditions stated at the end of 9.3 are satisfied. 20/5

The exception Program_Error is raised if all alternatives are closed and there is no else part. 21

NOTE A selective_accept can have several open delay_alternatives. A selective_accept can have several open accept_alternatives for the same entry. 22/5

Examples

Example of a task body with a selective accept: 23

```
task body Server is                                                    24
   Current_Work_Item : Work_Item;
begin
   loop
      select
         accept Next_Work_Item(WI : in Work_Item) do
            Current_Work_Item := WI;
         end;
         Process_Work_Item(Current_Work_Item);
      or
         accept Shut_Down;
         exit;          -- Premature shut down requested
      or
         terminate;     -- Normal shutdown at end of scope
      end select;
   end loop;
end Server;
```

9.7.2 Timed Entry Calls

1/5 A timed_entry_call issues an entry call that is cancelled if the call (or a requeue-with-abort of the call) is not selected before the expiration time is reached. A procedure call may appear rather than an entry call for cases where the procedure can be implemented by an entry.

Syntax

2
```
timed_entry_call ::=
  select
   entry_call_alternative
  or
   delay_alternative
  end select;
```

3/2
```
entry_call_alternative ::=
  procedure_or_entry_call [sequence_of_statements]
```

3.1/2
```
procedure_or_entry_call ::=
  procedure_call_statement | entry_call_statement
```

Legality Rules

3.2/2 If a procedure_call_statement is used for a procedure_or_entry_call, the *procedure*_name or *procedure*_prefix of the procedure_call_statement shall statically denote an entry renamed as a procedure or (a view of) a primitive subprogram of a limited interface whose first parameter is a controlling parameter (see 3.9.2).

Dynamic Semantics

4/2 For the execution of a timed_entry_call, the *entry*_name, *procedure*_name, or *procedure*_prefix, and any actual parameters are evaluated, as for a simple entry call (see 9.5.3) or procedure call (see 6.4). The expiration time (see 9.6) for the call is determined by evaluating the *delay*_expression of the delay_alternative. If the call is an entry call or a call on a procedure implemented by an entry, the entry call is then issued. Otherwise, the call proceeds as described in 6.4 for a procedure call, followed by the sequence_of_statements of the entry_call_alternative; the sequence_of_statements of the delay_alternative is ignored.

5 If the call is queued (including due to a requeue-with-abort), and not selected before the expiration time is reached, an attempt to cancel the call is made. If the call completes due to the cancellation, the optional sequence_of_statements of the delay_alternative is executed; if the entry call completes normally, the optional sequence_of_statements of the entry_call_alternative is executed.

Examples

6 *Example of a timed entry call:*

7
```
select
   Controller.Request(Medium)(Some_Item);
or
   delay 45.0;
   -- controller too busy, try something else
end select;
```

9.7.3 Conditional Entry Calls

A conditional_entry_call issues an entry call that is then cancelled if it is not selected immediately (or if a requeue-with-abort of the call is not selected immediately). A procedure call may appear rather than an entry call for cases where the procedure can be implemented by an entry. 1/5

Syntax

conditional_entry_call ::= 2
 select
  entry_call_alternative
 else
  sequence_of_statements
 end select;

Dynamic Semantics

The execution of a conditional_entry_call is defined to be equivalent to the execution of a timed_entry_call with a delay_alternative specifying an immediate expiration time and the same sequence_of_statements as given after the reserved word **else**. 3

NOTE A conditional_entry_call can briefly increase the Count attribute of the entry, even if the conditional call is not selected. 4/5

Examples

Example of a conditional entry call: 5

```
procedure Spin(R : in out Resource) is  -- see 9.4                    6/5
begin
   loop
      select
         R.Seize;
         return;
      else
         null;  -- busy waiting
      end select;
   end loop;
end;
```

9.7.4 Asynchronous Transfer of Control

An asynchronous select_statement provides asynchronous transfer of control upon completion of an entry call or the expiration of a delay. 1

Syntax

asynchronous_select ::= 2
 select
  triggering_alternative
 then abort
  abortable_part
 end select;

triggering_alternative ::= triggering_statement [sequence_of_statements] 3

triggering_statement ::= procedure_or_entry_call | delay_statement 4/2

5 abortable_part ::= sequence_of_statements

Dynamic Semantics

6/2 For the execution of an asynchronous_select whose triggering_statement is a procedure_or_entry_call, the *entry_*name, *procedure_*name, or *procedure_*prefix, and actual parameters are evaluated as for a simple entry call (see 9.5.3) or procedure call (see 6.4). If the call is an entry call or a call on a procedure implemented by an entry, the entry call is issued. If the entry call is queued (or requeued-with-abort), then the abortable_part is executed. If the entry call is selected immediately, and never requeued-with-abort, then the abortable_part is never started. If the call is on a procedure that is not implemented by an entry, the call proceeds as described in 6.4, followed by the sequence_of_statements of the triggering_alternative; the abortable_part is never started.

7 For the execution of an asynchronous_select whose triggering_statement is a delay_statement, the *delay_*expression is evaluated and the expiration time is determined, as for a normal delay_statement. If the expiration time has not already passed, the abortable_part is executed.

8 If the abortable_part completes and is left prior to completion of the triggering_statement, an attempt to cancel the triggering_statement is made. If the attempt to cancel succeeds (see 9.5.3 and 9.6), the asynchronous_select is complete.

9 If the triggering_statement completes other than due to cancellation, the abortable_part is aborted (if started but not yet completed — see 9.8). If the triggering_statement completes normally, the optional sequence_of_statements of the triggering_alternative is executed after the abortable_part is left.

Examples

10 *Example of a main command loop for a command interpreter:*

11
```
loop
   select
      Terminal.Wait_For_Interrupt;
      Put_Line("Interrupted");
   then abort
      -- This will be abandoned upon terminal interrupt
      Put_Line("-> ");
      Get_Line(Command, Last);
      Process_Command(Command(1..Last));
   end select;
end loop;
```

12 *Example of a time-limited calculation:*

13/5
```
select
   delay 5.0;
   Put_Line("Calculation does not converge");
then abort
   -- This calculation is expected to finish in 5.0 seconds;
   -- if not, it is assumed to diverge.
   Horribly_Complicated_Recursive_Function(X, Y);
end select;
```

14/5 Note that these examples presume that there are abort completion points (see 9.8) within the execution of the abortable_part.

9.8 Abort of a Task - Abort of a Sequence of Statements

An abort_statement causes one or more tasks to become abnormal, thus preventing any further interaction with such tasks. The completion of the triggering_statement of an asynchronous_select causes a sequence_of_statements to be aborted. 1

Syntax

abort_statement ::= **abort** *task_*name {, *task_*name}; 2

Name Resolution Rules

Each *task_*name is expected to be of any task type; each can be of a different task type. 3/5

Dynamic Semantics

For the execution of an abort_statement, the given *task_*names are evaluated in an arbitrary order. Each named task is then *aborted*, which consists of making the task *abnormal* and aborting the execution of the corresponding task_body, unless it is already completed. 4

When the execution of a construct is *aborted* (including that of a task_body or of a sequence_of_-statements), the execution of every construct included within the aborted execution is also aborted, except for executions included within the execution of an *abort-deferred* operation; the execution of an abort-deferred operation continues to completion without being affected by the abort; the following are the abort-deferred operations: 5

- a protected action; 6
- waiting for an entry call to complete (after having initiated the attempt to cancel it — see below); 7
- waiting for the termination of dependent tasks; 8
- the execution of an Initialize procedure as the last step of the default initialization of a controlled object; 9
- the execution of a Finalize procedure as part of the finalization of a controlled object; 10
- an assignment operation to an object with a controlled part. 11

The last three of these are discussed further in 7.6. 12

When a master is aborted, all tasks that depend on that master are aborted. 13

The order in which tasks become abnormal as the result of an abort_statement or the abort of a sequence_of_statements is not specified by the language. 14

If the execution of an entry call is aborted, an immediate attempt is made to cancel the entry call (see 9.5.3). If the execution of a construct is aborted at a time when the execution is blocked, other than for an entry call, at a point that is outside the execution of an abort-deferred operation, then the execution of the construct completes immediately. For an abort due to an abort_statement, these immediate effects occur before the execution of the abort_statement completes. Other than for these immediate cases, the execution of a construct that is aborted does not necessarily complete before the abort_statement completes. However, the execution of the aborted construct completes no later than its next *abort completion point* (if any) that occurs outside of an abort-deferred operation; the following are abort completion points for an execution: 15

- the point where the execution initiates the activation of another task; 16

17 • the end of the activation of a task;

17.1/5 • a point within a parallel construct where a new logical thread of control is created;

17.2/5 • the end of a parallel construct;

18 • the start or end of the execution of an entry call, accept_statement, delay_statement, or abort_statement;

19 • the start of the execution of a select_statement, or of the sequence_of_statements of an exception_handler.

Bounded (Run-Time) Errors

20/5 An attempt to execute an asynchronous_select as part of the execution of an abort-deferred operation is a bounded error. Similarly, an attempt to create a task that depends on a master that is included entirely within the execution of an abort-deferred operation is a bounded error. In both cases, Program_Error is raised if the error is detected by the implementation; otherwise, the operations proceed as they would outside an abort-deferred operation, except that an abort of the abortable_part or the created task does not necessarily have an effect.

Erroneous Execution

21 If an assignment operation completes prematurely due to an abort, the assignment is said to be *disrupted*; the target of the assignment or its parts can become abnormal, and certain subsequent uses of the object can be erroneous, as explained in 13.9.1.

22/5 NOTE 1 An abort_statement is best used only in situations requiring unconditional termination.

23 NOTE 2 A task is allowed to abort any task it can name, including itself.

24/5 NOTE 3 Additional requirements associated with abort are given in D.6.

9.9 Task and Entry Attributes

Dynamic Semantics

1 For a prefix T that is of a task type (after any implicit dereference), the following attributes are defined:

2 T'Callable Yields the value True when the task denoted by T is *callable*, and False otherwise; a task is callable unless it is completed or abnormal. The value of this attribute is of the predefined type Boolean.

3 T'Terminated Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean.

4 For a prefix E that denotes an entry of a task or protected unit, the following attribute is defined. This attribute is only allowed within the body of the task or protected unit, but excluding, in the case of an entry of a task unit, within any program unit that is, itself, inner to the body of the task unit.

5 E'Count Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type *universal_integer*.

6 NOTE 1 For the Count attribute, the entry can be either a single entry or an entry of a family. The name of the entry or entry family can be either a direct_name or an expanded name.

7/5 NOTE 2 Within task units, by interrogating the attribute E'Count an algorithm can allow for the increase of the value of this attribute for incoming entry calls, and its decrease, for example with timed_entry_calls. A conditional_entry_call can also briefly increase this value, even if the conditional call is not accepted.

8/5 NOTE 3 Within protected units, by interrogating the attribute E'Count in the entry_barrier for the entry E an algorithm can allow for the evaluation of the condition of the barrier both before and after queuing a given caller.

9.10 Shared Variables

Static Semantics

If two different objects, including nonoverlapping parts of the same object, are *independently addressable*, they can be manipulated concurrently by two different logical threads of control without synchronization, unless both are subcomponents of the same full access object, and either is nonatomic (see C.6). Any two nonoverlapping objects are independently addressable if either object is specified as independently addressable (see C.6). Otherwise, two nonoverlapping objects are independently addressable except when they are both parts of a composite object for which a nonconfirming value is specified for any of the following representation aspects: (record) Layout, Component_Size, Pack, Atomic, or Convention; in this case it is unspecified whether the parts are independently addressable. 1/5

Dynamic Semantics

Separate logical threads of control normally proceed independently and concurrently with one another. However, task interactions can be used to synchronize the actions of two or more logical threads of control to allow, for example, meaningful communication by the direct updating and reading of variables shared between them. The actions of two different logical threads of control are synchronized in this sense when an action of one *signals* an action of the other; an action A1 is defined to signal an action A2 under the following circumstances: 2/5

- If A1 and A2 are part of the execution of the same task, and the language rules require A1 to be performed before A2; 3
- If A1 is the action of an activator that initiates the activation of a task, and A2 is part of the execution of the task that is activated; 4
- If A1 is part of the activation of a task, and A2 is the action of waiting for completion of the activation; 5
- If A1 is part of the execution of a task, and A2 is the action of waiting for the termination of the task; 6
- If A1 is the termination of a task T, and A2 is either an evaluation of the expression T'Terminated that results in True, or a call to Ada.Task_Identification.Is_Terminated with an actual parameter that identifies T and a result of True (see C.7.1); 6.1/3
- If A1 is the action of issuing an entry call, and A2 is part of the corresponding execution of the appropriate entry_body or accept_statement; 7/3
- If A1 is part of the execution of an accept_statement or entry_body, and A2 is the action of returning from the corresponding entry call; 8
- If A1 is part of the execution of a protected procedure body or entry_body for a given protected object, and A2 is part of a later execution of an entry_body for the same protected object; 9
- If A1 signals some action that in turn signals A2. 10

Action A1 is defined to *potentially signal* action A2 if A1 signals A2, if action A1 and A2 occur as part of the execution of the same logical thread of control, and the language rules permit action A1 to precede action A2, or if action A1 potentially signals some action that in turn potentially signals A2. 10.1/5

Two actions are defined to be *sequential* if one of the following is true: 11/5

- One action signals the other; 12
- Both actions occur as part of the execution of the same logical thread of control; 13/5

14/5 • Both actions occur as part of protected actions on the same protected object, and at least one of the actions is part of a call on an exclusive protected operation of the protected object.

15/3 Aspect Atomic or aspect Atomic_Components may also be specified to ensure that certain reads and updates are sequential — see C.6.

16/5 Two actions that are not sequential are defined to be *concurrent* actions.

17/5 Two actions are defined to *conflict* if one action assigns to an object, and the other action reads or assigns to a part of the same object (or of a neighboring object if the two are not independently addressable). The action comprising a call on a subprogram or an entry is defined to *potentially conflict* with another action if the Global aspect (or Global'Class aspect in the case of a dispatching call) of the called subprogram or entry is such that a conflicting action would be possible during the execution of the call. Similarly, two calls are considered to potentially conflict if they each have Global (or Global'Class in the case of a dispatching call) aspects such that conflicting actions would be possible during the execution of the calls. Finally, two actions that conflict are also considered to potentially conflict.

18/5 A *synchronized* object is an object of a task or protected type, an atomic object (see C.6), a suspension object (see D.10), or a synchronous barrier (see D.10.1). Operations on such objects are necessarily sequential with respect to one another, and hence are never considered to conflict.

Erroneous Execution

19/5 The execution of two concurrent actions is erroneous if the actions make conflicting uses of a shared variable (or neighboring variables that are not independently addressable).

9.10.1 Conflict Check Policies

1/5 This subclause determines what checks are performed relating to possible concurrent conflicting actions (see 9.10).

Syntax

2/5 The form of a pragma Conflict_Check_Policy is as follows:

3/5 **pragma** Conflict_Check_Policy (*policy_*identifier[, *policy_*identifier]);

4/5 A pragma Conflict_Check_Policy is allowed only immediately within a declarative_part, a package_specification, or as a configuration pragma.

Legality Rules

5/5 Each *policy_*identifier shall be one of No_Parallel_Conflict_Checks, Known_Parallel_Conflict_Checks, All_Parallel_Conflict_Checks, No_Tasking_Conflict_Checks, Known_Tasking_Conflict_Checks, All_Tasking_Conflict_Checks, No_Conflict_Checks, Known_Conflict_Checks, All_Conflict_Checks, or an implementation-defined conflict check policy. If two *policy_*identifiers are given, one shall include the word Parallel and one shall include the word Tasking. If only one *policy_*identifier is given, it shall not include the word Parallel or Tasking.

6/5 A pragma Conflict_Check_Policy given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a pragma Conflict_Check_Policy given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies.

7/5 If a pragma Conflict_Check_Policy applies to a generic_instantiation, then the pragma Conflict_Check_Policy applies to the entire instance.

If multiple Conflict_Check_Policy pragmas apply to a given construct, the conflict check policy is determined by the one in the innermost enclosing region. If no Conflict_Check_Policy pragma applies to a construct, the policy is (All_Parallel_Conflict_Checks, No_Tasking_Conflict_Checks) (see below). 8/5

Certain potentially conflicting actions are disallowed according to which conflict check policies apply at the place where the action or actions occur, as follows: 9/5

No_Parallel_Conflict_Checks 10/5

This policy imposes no restrictions on concurrent actions arising from parallel constructs.

No_Tasking_Conflict_Checks 11/5

This policy imposes no restrictions on concurrent actions arising from tasking constructs.

Known_Parallel_Conflict_Checks 12/5

If this policy applies to two concurrent actions appearing within parallel constructs, they are disallowed if they are known to denote the same object (see 6.4.1) with uses that conflict. For the purposes of this check, any parallel loop may be presumed to involve multiple concurrent iterations. Also, for the purposes of deciding whether two actions are concurrent, it is enough for the logical threads of control in which they occur to be concurrent at any point in their execution, unless all of the following are true:

- the shared object is volatile; 13/5
- the two logical threads of control are both known to also refer to a shared synchronized object; and 14/5
- each thread whose potentially conflicting action updates the shared volatile object, also updates this shared synchronized object. 15/5

Known_Tasking_Conflict_Checks 16/5

If this policy applies to two concurrent actions appearing within the same compilation unit, at least one of which appears within a task body but not within a parallel construct, they are disallowed if they are known to denote the same object (see 6.4.1) with uses that conflict, and neither potentially signals the other (see 9.10). For the purposes of this check, any named task type may be presumed to have multiple instances. Also, for the purposes of deciding whether two actions are concurrent, it is enough for the tasks in which they occur to be concurrent at any point in their execution, unless all of the following are true:

- the shared object is volatile; 17/5
- the two tasks are both known to also refer to a shared synchronized object; and 18/5
- each task whose potentially conflicting action updates the shared volatile object, also updates this shared synchronized object. 19/5

All_Parallel_Conflict_Checks 20/5

This policy includes the restrictions imposed by the Known_Parallel_Conflict_Checks policy, and in addition disallows a parallel construct from reading or updating a variable that is global to the construct, unless it is a synchronized object, or unless the construct is a parallel loop, and the global variable is a part of a component of an array denoted by an indexed component with at least one index expression that statically denotes the loop parameter of the loop_parameter_specification or the chunk parameter of the parallel loop.

All_Tasking_Conflict_Checks 21/5

This policy includes the restrictions imposed by the Known_Tasking_Conflict_Checks policy, and in addition disallows a task body from reading or updating a variable that is global to the task body, unless it is a synchronized object.

22/5 No_Conflict_Checks, Known_Conflict_Checks, All_Conflict_Checks
These are shorthands for (No_Parallel_Conflict_Checks, No_Tasking_Conflict_Checks), (Known_Parallel_Conflict_Checks, Known_Tasking_Conflict_Checks), and (All_Parallel_Conflict_Checks, All_Tasking_Conflict_Checks), respectively.

Static Semantics

23/5 For a subprogram, the following language-defined representation aspect may be specified:

24/5 Parallel_Calls
The Parallel_Calls aspect is of type Boolean. The specified value shall be static. The Parallel_Calls aspect of an inherited primitive subprogram is True if Parallel_Calls is True either for the corresponding subprogram of the progenitor type or for any other inherited subprogram that it overrides. If not specified or inherited as True, the Parallel_Calls aspect of a subprogram is False.

25/5 Specifying the Parallel_Calls aspect to be True for a subprogram indicates that the subprogram can be safely called in parallel. Conflict checks (if required by the Conflict_Check_Policy in effect) are made on the subprogram assuming that multiple concurrent calls exist. Such checks can then be omitted on a call of the subprogram in a parallel iteration context.

Implementation Permissions

26/5 When the conflict check policy Known_Parallel_Conflict_Checks or All_Parallel_Conflict_Checks applies, the implementation may disallow two concurrent actions appearing within parallel constructs if the implementation can prove they will at run-time denote the same object with uses that conflict. Similarly, when the conflict check policy Known_Tasking_Conflict_Checks or All_Tasking_Conflict_Checks applies, the implementation may disallow two concurrent actions, at least one of which appears within a task body but not within a parallel construct, if the implementation can prove they will at run-time denote the same object with uses that conflict.

9.11 Example of Tasking and Synchronization

Examples

1/5 The following example defines a buffer protected object to smooth variations between the speed of output of a producing task and the speed of input of some consuming task. For instance, the producing task can have the following structure:

2
```
task Producer;
```

3/2
```
task body Producer is
   Person : Person_Name; -- see 3.10.1
begin
   loop
      ... -- simulate arrival of the next customer
      Buffer.Append_Wait(Person);
      exit when Person = null;
   end loop;
end Producer;
```

4/5 and the consuming task can have the following structure:

5
```
task Consumer;
```

```
task body Consumer is                                                    6/2
   Person : Person_Name;
begin
   loop
      Buffer.Remove_First_Wait(Person);
      exit when Person = null;
      ... -- simulate serving a customer
   end loop;
end Consumer;
```

The buffer object contains an internal array of person names managed in a round-robin fashion. The array has two indices, an In_Index denoting the index for the next input person name and an Out_Index denoting the index for the next output person name. 7/2

The Buffer is defined as an extension of the Synchronized_Queue interface (see 3.9.4), and as such promises to implement the abstraction defined by that interface. By doing so, the Buffer can be passed to the Transfer class-wide operation defined for objects of a type covered by Queue'Class. 7.1/2

```
type Person_Name_Array is array (Positive range <>)                      7.2/5
   of Person_Name;  -- see 3.10.1

protected Buffer is new Synchronized_Queue with  -- see 3.9.4            8/2
   entry Append_Wait(Person : in Person_Name);
   entry Remove_First_Wait(Person : out Person_Name);
   function Cur_Count return Natural;
   function Max_Count return Natural;
   procedure Append(Person : in Person_Name);
   procedure Remove_First(Person : out Person_Name);
private
   Pool      : Person_Name_Array(1 .. 100);
   Count     : Natural := 0;
   In_Index, Out_Index : Positive := 1;
end Buffer;

protected body Buffer is                                                 9/2
   entry Append_Wait(Person : in Person_Name)
      when Count < Pool'Length is
   begin
      Append(Person);
   end Append_Wait;

   procedure Append(Person : in Person_Name) is                          9.1/2
   begin
      if Count = Pool'Length then
         raise Queue_Error with "Buffer Full";  -- see 11.3
      end if;
      Pool(In_Index) := Person;
      In_Index       := (In_Index mod Pool'Length) + 1;
      Count          := Count + 1;
   end Append;

   entry Remove_First_Wait(Person : out Person_Name)                     10/2
      when Count > 0 is
   begin
      Remove_First(Person);
   end Remove_First_Wait;

   procedure Remove_First(Person : out Person_Name) is                   11/2
   begin
      if Count = 0 then
         raise Queue_Error with "Buffer Empty"; -- see 11.3
      end if;
      Person    := Pool(Out_Index);
      Out_Index := (Out_Index mod Pool'Length) + 1;
      Count     := Count - 1;
   end Remove_First;
```

12/2
```
        function Cur_Count return Natural is
        begin
            return Buffer.Count;
        end Cur_Count;
```
13/2
```
        function Max_Count return Natural is
        begin
            return Pool'Length;
        end Max_Count;
    end Buffer;
```

10 Program Structure and Compilation Issues

The overall structure of programs and the facilities for separate compilation are described in this clause. A *program* is a set of *partitions*, each of which may execute in a separate address space, possibly on a separate computer. 1/3

As explained below, a partition is constructed from *library units*. Syntactically, the declaration of a library unit is a library_item, as is the body of a library unit. An implementation may support a concept of a *program library* (or simply, a "library"), which contains library_items and their subunits. Library units may be organized into a hierarchy of children, grandchildren, and so on. 2

This clause has two subclauses: 10.1, "Separate Compilation" discusses compile-time issues related to separate compilation. 10.2, "Program Execution" discusses issues related to what is traditionally known as "link time" and "run time" — building and executing partitions. 3/3

10.1 Separate Compilation

A *program unit* is either a package, a task unit, a protected unit, a protected entry, a generic unit, or an explicitly declared subprogram other than an enumeration literal. Certain kinds of program units can be separately compiled. Alternatively, they can appear physically nested within other program units. 1

The text of a program can be submitted to the compiler in one or more compilations. Each compilation is a succession of compilation_units. A compilation_unit contains either the declaration, the body, or a renaming of a program unit. The representation for a compilation is implementation-defined. 2

A library unit is a separately compiled program unit, and is a package, subprogram, or generic unit. Library units may have other (logically nested) library units as children, and may have other program units physically nested within them. A root library unit, together with its children and grandchildren and so on, form a *subsystem*. 3

Implementation Permissions

An implementation may impose implementation-defined restrictions on compilations that contain multiple compilation_units. 4

10.1.1 Compilation Units - Library Units

A library_item is a compilation unit that is the declaration, body, or renaming of a library unit. Each library unit (except Standard) has a *parent unit*, which is a library package or generic library package. A library unit is a *child* of its parent unit. The *root* library units are the children of the predefined library package Standard. 1

Syntax

```
compilation ::= {compilation_unit}                                    2

compilation_unit ::=                                                  3
   context_clause library_item
 | context_clause subunit
```

4 library_item ::= [**private**] library_unit_declaration
| library_unit_body
| [**private**] library_unit_renaming_declaration

5 library_unit_declaration ::=
subprogram_declaration | package_declaration
| generic_declaration | generic_instantiation

6 library_unit_renaming_declaration ::=
package_renaming_declaration
| generic_renaming_declaration
| subprogram_renaming_declaration

7 library_unit_body ::= subprogram_body | package_body

8 parent_unit_name ::= name

8.1/2 An overriding_indicator is not allowed in a subprogram_declaration, generic_instantiation, or subprogram_renaming_declaration that declares a library unit.

9 A *library unit* is a program unit that is declared by a library_item. When a program unit is a library unit, the prefix "library" is used to refer to it (or "generic library" if generic), as well as to its declaration and body, as in "library procedure", "library package_body", or "generic library package". The term *compilation unit* is used to refer to a compilation_unit. When the meaning is clear from context, the term is also used to refer to the library_item of a compilation_unit or to the proper_body of a subunit (that is, the compilation_unit without the context_clause and the **separate** (parent_unit_name)).

10 The *parent declaration* of a library_item (and of the library unit) is the declaration denoted by the parent_unit_name, if any, of the defining_program_unit_name of the library_item. If there is no parent_unit_name, the parent declaration is the declaration of Standard, the library_item is a *root* library_item, and the library unit (renaming) is a *root* library unit (renaming). The declaration and body of Standard itself have no parent declaration. The *parent unit* of a library_item or library unit is the library unit declared by its parent declaration.

11 The children of a library unit occur immediately within the declarative region of the declaration of the library unit. The *ancestors* of a library unit are itself, its parent, its parent's parent, and so on. (Standard is an ancestor of every library unit.) The *descendant* relation is the inverse of the ancestor relation.

12 A library_unit_declaration or a library_unit_renaming_declaration is *private* if the declaration is immediately preceded by the reserved word **private**; it is otherwise *public*. A library unit is private or public according to its declaration. The *public descendants* of a library unit are the library unit itself, and the public descendants of its public children. Its other descendants are *private descendants*.

12.1/2 For each library package_declaration in the environment, there is an implicit declaration of a *limited view* of that library package. The limited view of a package contains:

12.2/3 - For each package_declaration occurring immediately within the visible part, a declaration of the limited view of that package, with the same defining_program_unit_name.

12.3/3 - For each type_declaration occurring immediately within the visible part that is not an incomplete_type_declaration, an incomplete view of the type with no discriminant_part; if the type_declaration is tagged, then the view is a tagged incomplete view.

12.4/2 The limited view of a library package_declaration is private if that library package_declaration is immediately preceded by the reserved word **private**.

There is no syntax for declaring limited views of packages, because they are always implicit. The implicit declaration of a limited view of a library package is not the declaration of a library unit (the library package_declaration is); nonetheless, it is a library_item. The implicit declaration of the limited view of a library package forms an (implicit) compilation unit whose context_clause is empty. 12.5/2

A library package_declaration is the completion of the declaration of its limited view. 12.6/2

Legality Rules

The parent unit of a library_item shall be a library package or generic library package. 13

If a defining_program_unit_name of a given declaration or body has a parent_unit_name, then the given declaration or body shall be a library_item. The body of a program unit shall be a library_item if and only if the declaration of the program unit is a library_item. In a library_unit_renaming_declaration, the (old) name shall denote a library_item. 14

A parent_unit_name (which can be used within a defining_program_unit_name of a library_item and in the **separate** clause of a subunit), and each of its prefixes, shall not denote a renaming_declaration. On the other hand, a name that denotes a library_unit_renaming_declaration is allowed in a nonlimited_with_clause and other places where the name of a library unit is allowed. 15/2

If a library package is an instance of a generic package, then every child of the library package shall either be itself an instance or be a renaming of a library unit. 16

A child of a generic library package shall either be itself a generic unit or be a renaming of some other child of the same generic unit. 17/3

A child of a parent generic package shall be instantiated or renamed only within the declarative region of the parent generic. 18

For each child *C* of some parent generic package *P*, there is a corresponding declaration *C* nested immediately within each instance of *P*. For the purposes of this rule, if a child *C* itself has a child *D*, each corresponding declaration for *C* has a corresponding child *D*. The corresponding declaration for a child within an instance is visible only within the scope of a with_clause that mentions the (original) child generic unit. 19/2

A library subprogram shall not override a primitive subprogram. 20

The defining name of a function that is a compilation unit shall not be an operator_symbol. 21

Static Semantics

A subprogram_renaming_declaration that is a library_unit_renaming_declaration is a renaming-as-declaration, not a renaming-as-body. 22

There are two kinds of dependences among compilation units: 23

- The *semantic dependences* (see below) are the ones necessary to check the compile-time rules across compilation unit boundaries; a compilation unit depends semantically on the other compilation units necessary to determine its legality. The visibility rules are based on the semantic dependences. 24/5
- The *elaboration dependences* (see 10.2) determine the order of elaboration of library_items. 25

A library_item depends semantically upon its parent declaration. A subunit depends semantically upon its parent body. A library_unit_body depends semantically upon the corresponding library_unit_declaration, if any. The declaration of the limited view of a library package depends semantically upon the declaration 26/2

of the limited view of its parent. The declaration of a library package depends semantically upon the declaration of its limited view. A compilation unit depends semantically upon each library_item mentioned in a with_clause of the compilation unit. In addition, if a given compilation unit contains an attribute_reference of a type defined in another compilation unit, then the given compilation unit depends semantically upon the other compilation unit. The semantic dependence relationship is transitive.

Dynamic Semantics

26.1/2 The elaboration of the declaration of the limited view of a package has no effect.

27/5 NOTE 1 A simple program can consist of a single compilation unit. A compilation can have no compilation units; for example, its text can consist of pragmas.

28 NOTE 2 The designator of a library function cannot be an operator_symbol, but a nonlibrary renaming_declaration is allowed to rename a library function as an operator. Within a partition, two library subprograms are required to have distinct names and hence cannot overload each other. However, renaming_declarations are allowed to define overloaded names for such subprograms, and a locally declared subprogram is allowed to overload a library subprogram. The expanded name Standard.L can be used to denote a root library unit L (unless the declaration of Standard is hidden) since root library unit declarations occur immediately within the declarative region of package Standard.

Examples

29 *Examples of library units:*

30
```
package Rational_Numbers.IO is   -- public child of Rational_Numbers, see 7.1
   procedure Put(R : in  Rational);
   procedure Get(R : out Rational);
end Rational_Numbers.IO;
```

31
```
private procedure Rational_Numbers.Reduce(R : in out Rational);
                                 -- private child of Rational_Numbers
```

32
```
with Rational_Numbers.Reduce;    -- refer to a private child
package body Rational_Numbers is
   ...
end Rational_Numbers;
```

33
```
with Rational_Numbers.IO; use Rational_Numbers;
with Ada.Text_io;                -- see A.10
procedure Main is                -- a root library procedure
   R : Rational;
begin
   R := 5/3;                     -- construct a rational number, see 7.1
   Ada.Text_IO.Put("The answer is: ");
   IO.Put(R);
   Ada.Text_IO.New_Line;
end Main;
```

34
```
with Rational_Numbers.IO;
package Rational_IO renames Rational_Numbers.IO;
                                 -- a library unit renaming declaration
```

35 Each of the above library_items can be submitted to the compiler separately.

10.1.2 Context Clauses - With Clauses

1 A context_clause is used to specify the library_items whose names are needed within a compilation unit.

Syntax

2 context_clause ::= {context_item}

3 context_item ::= with_clause | use_clause

4/2 with_clause ::= limited_with_clause | nonlimited_with_clause

limited_with_clause ::= **limited** [**private**] **with** *library_unit*_name {, *library_unit*_name}; 4.1/2

nonlimited_with_clause ::= [**private**] **with** *library_unit*_name {, *library_unit*_name}; 4.2/2

Name Resolution Rules

The *scope* of a with_clause that appears on a library_unit_declaration or library_unit_renaming_-declaration consists of the entire declarative region of the declaration, which includes all children and subunits. The scope of a with_clause that appears on a body consists of the body, which includes all subunits. 5

A library_item (and the corresponding library unit) is *named* in a with_clause if it is denoted by a *library_unit*_name in the with_clause. A library_item (and the corresponding library unit) is *mentioned* in a with_clause if it is named in the with_clause or if it is denoted by a prefix in the with_clause. 6/2

Outside its own declarative region, the declaration or renaming of a library unit can be visible only within the scope of a with_clause that mentions it. The visibility of the declaration or renaming of a library unit otherwise follows from its placement in the environment. 7

Legality Rules

If a with_clause of a given compilation_unit mentions a private child of some library unit, then the given compilation_unit shall be one of: 8/2

- the declaration, body, or subunit of a private descendant of that library unit; 9/2
- the body or subunit of a public descendant of that library unit, but not a subprogram body acting as a subprogram declaration (see 10.1.4); or 10/2
- the declaration of a public descendant of that library unit, in which case the with_clause shall include the reserved word **private**. 11/2

A name denoting a library_item (or the corresponding declaration for a child of a generic within an instance — see 10.1.1), if it is visible only due to being mentioned in one or more with_clauses of a unit *U* that include the reserved word **private**, shall appear only within: 12/5

- a private part; 13/2
- a body of a public descendant of *U*, but not within the subprogram_specification of a body of a subprogram that is a public descendant of *U*; 14/5
- a private descendant of *U* or its body; or 15/5
- a pragma within a context clause. 16/2

A library_item mentioned in a limited_with_clause shall be the implicit declaration of the limited view of a library package, not the declaration of a subprogram, generic unit, generic instance, or a renaming. 17/2

A limited_with_clause shall not appear on a library_unit_body, subunit, or library_unit_renaming_-declaration. 18/2

A limited_with_clause that names a library package shall not appear: 19/2

- in the context_clause for the explicit declaration of the named library package or any of its descendants; 20/3
- within a context_clause for a library_item that is within the scope of a nonlimited_with_clause that mentions the same library package; or 21/3
- within a context_clause for a library_item that is within the scope of a use_clause that names an entity declared within the declarative region of the library package. 22/3

23/2 NOTE A library_item mentioned in a nonlimited_with_clause of a compilation unit is visible within the compilation unit and hence acts just like an ordinary declaration. Thus, within a compilation unit that mentions its declaration, the name of a library package can be given in use_clauses and can be used to form expanded names, a library subprogram can be called, and instances of a generic library unit can be declared. If a child of a parent generic package is mentioned in a nonlimited_with_clause, then the corresponding declaration nested within each visible instance is visible within the compilation unit. Similarly, a library_item mentioned in a limited_with_clause of a compilation unit is visible within the compilation unit and thus can be used to form expanded names.

Examples

24/5 *Examples of use of with clauses, limited with clauses, and private with clauses:*

25/2
```
package Office is
end Office;
```

26/2
```
with Ada.Strings.Unbounded;
package Office.Locations is
   type Location is new Ada.Strings.Unbounded.Unbounded_String;
end Office.Locations;
```

27/2
```
limited with Office.Departments;   -- types are incomplete
private with Office.Locations;     -- only visible in private part
package Office.Employees is
   type Employee is private;
```

28/2
```
   function Dept_Of(Emp : Employee) return access Departments.Department;
   procedure Assign_Dept(Emp  : in out Employee;
                         Dept : access Departments.Department);
```

29/2
```
   ...
private
   type Employee is
      record
         Dept : access Departments.Department;
         Loc : Locations.Location;
         ...
      end record;
end Office.Employees;
```

30/5
```
limited with Office.Employees;
package Office.Departments is
   type Department is ...;
```

31/2
```
   function Manager_Of(Dept : Department) return access Employees.Employee;
   procedure Assign_Manager(Dept : in out Department;
                            Mgr  : access Employees.Employee);
   ...
end Office.Departments;
```

32/5 *The* limited_with_clause *can be used to support mutually dependent abstractions that are split across multiple packages. In this case, an employee is assigned to a department, and a department has a manager who is an employee. If a* with_clause *with the reserved word* ***private*** *appears on one library unit and mentions a second library unit, it provides visibility to the second library unit, but restricts that visibility to the private part and body of the first unit. The compiler checks that no use is made of the second unit in the visible part of the first unit.*

10.1.3 Subunits of Compilation Units

1 Subunits are like child units, with these (important) differences: subunits support the separate compilation of bodies only (not declarations); the parent contains a body_stub to indicate the existence and place of each of its subunits; declarations appearing in the parent's body can be visible within the subunits.

Syntax

```
body_stub ::=                                                          2
   subprogram_body_stub | package_body_stub
 | task_body_stub | protected_body_stub

subprogram_body_stub ::=                                               3/3
   [overriding_indicator]
   subprogram_specification is separate
     [aspect_specification];

package_body_stub ::=                                                  4/3
  package body defining_identifier is separate
    [aspect_specification];

task_body_stub ::=                                                     5/3
  task body defining_identifier is separate
    [aspect_specification];

protected_body_stub ::=                                                6/3
  protected body defining_identifier is separate
    [aspect_specification];

subunit ::= separate (parent_unit_name) proper_body                    7
```

Legality Rules

The *parent body* of a subunit is the body of the program unit denoted by its parent_unit_name. The term *subunit* is used to refer to a subunit and also to the proper_body of a subunit. The *subunits of a program unit* include any subunit that names that program unit as its parent, as well as any subunit that names such a subunit as its parent (recursively). 8/2

The parent body of a subunit shall be present in the current environment, and shall contain a corresponding body_stub with the same defining_identifier as the subunit. 9

A package_body_stub shall be the completion of a package_declaration or generic_package_declaration; a task_body_stub shall be the completion of a task declaration; a protected_body_stub shall be the completion of a protected declaration. 10/3

In contrast, a subprogram_body_stub can be defined without it being the completion of a previous declaration, in which case the _stub declares the subprogram. If the _stub is a completion, it shall be the completion of a subprogram_declaration or generic_subprogram_declaration. The profile of a subprogram_body_stub that completes a declaration shall conform fully to that of the declaration. 11/5

A subunit that corresponds to a body_stub shall be of the same kind (package_, subprogram_, task_, or protected_) as the body_stub. The profile of a subprogram_body subunit shall be fully conformant to that of the corresponding body_stub. 12

A body_stub shall appear immediately within the declarative_part of a compilation unit body. This rule does not apply within an instance of a generic unit. 13

The defining_identifiers of all body_stubs that appear immediately within a particular declarative_part shall be distinct. 14

Post-Compilation Rules

For each body_stub, there shall be a subunit containing the corresponding proper_body. 15

16 NOTE The rules in 10.1.4, “The Compilation Process” say that a body_stub is equivalent to the corresponding proper_body. This implies:

17 • Visibility within a subunit is the visibility that would be obtained at the place of the corresponding body_stub (within the parent body) if the context_clause of the subunit were appended to that of the parent body.

18 • The effect of the elaboration of a body_stub is to elaborate the subunit.

Examples

19/5 *Example that defines package Parent without subunits:*

20
```
package Parent is
    procedure Inner;
end Parent;
```

21
```
with Ada.Text_IO;
package body Parent is
    Variable : String := "Hello, there.";
    procedure Inner is
    begin
        Ada.Text_IO.Put_Line(Variable);
    end Inner;
end Parent;
```

22/5 *Example showing how the body of procedure Inner can be turned into a subunit by rewriting the package body as follows (with the declaration of Parent remaining the same):*

23
```
package body Parent is
    Variable : String := "Hello, there.";
    procedure Inner is separate;
end Parent;
```

24
```
with Ada.Text_IO;
separate(Parent)
procedure Inner is
begin
    Ada.Text_IO.Put_Line(Variable);
end Inner;
```

10.1.4 The Compilation Process

1 Each compilation unit submitted to the compiler is compiled in the context of an *environment* declarative_part (or simply, an *environment*), which is a conceptual declarative_part that forms the outermost declarative region of the context of any compilation. At run time, an environment forms the declarative_part of the body of the environment task of a partition (see 10.2, “Program Execution”).

2/5 The declarative_items of the environment are library_items appearing in an order such that there are no forward semantic dependences. Each included subunit occurs in place of the corresponding stub. The visibility rules apply as if the environment were the outermost declarative region, except that with_clauses are necessary to make declarations of library units visible (see 10.1.2).

3/2 The mechanisms for creating an environment and for adding and replacing compilation units within an environment are implementation defined. The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment are implementation defined.

Name Resolution Rules

4/3 If a library_unit_body that is a subprogram_body is submitted to the compiler, it is interpreted only as a completion if a library_unit_declaration with the same defining_program_unit_name already exists in the environment for a subprogram other than an instance of a generic subprogram or for a generic subprogram (even if the profile of the body is not type conformant with that of the declaration); otherwise, the subprogram_body is interpreted as both the declaration and body of a library subprogram.

Legality Rules

When a compilation unit is compiled, all compilation units upon which it depends semantically shall already exist in the environment; the set of these compilation units shall be *consistent* in the sense that the new compilation unit shall not semantically depend (directly or indirectly) on two different versions of the same compilation unit, nor on an earlier version of itself. 5

Implementation Permissions

The implementation may require that a compilation unit be legal before it can be mentioned in a limited_with_clause or it can be inserted into the environment. 6/2

When a compilation unit that declares or renames a library unit is added to the environment, the implementation may remove from the environment any preexisting library_item or subunit with the same full expanded name. When a compilation unit that is a subunit or the body of a library unit is added to the environment, the implementation may remove from the environment any preexisting version of the same compilation unit. When a compilation unit that contains a body_stub is added to the environment, the implementation may remove any preexisting library_item or subunit with the same full expanded name as the body_stub. When a given compilation unit is removed from the environment, the implementation may also remove any compilation unit that depends semantically upon the given one. If the given compilation unit contains the body of a subprogram for which aspect Inline is True, the implementation may also remove any compilation unit containing a call to that subprogram. 7/3

NOTE 1 The rules of the language are enforced across compilation and compilation unit boundaries, just as they are enforced within a single compilation unit. 8

NOTE 2 An implementation can support a concept of a *library*, which contains library_items. If multiple libraries are supported, the implementation can document how a single environment is constructed when a compilation unit is submitted to the compiler. Naming conflicts between different libraries can, for example, be resolved by treating each library as the root of a hierarchy of child library units. 9/5

NOTE 3 A compilation unit containing an instantiation of a separately compiled generic unit does not semantically depend on the body of the generic unit. Therefore, replacing the generic body in the environment does not result in the removal of the compilation unit containing the instantiation. 10

10.1.5 Pragmas and Program Units

This subclause discusses pragmas related to compilations. 1/5

Paragraphs 2 through 7 were moved to Annex J, "Obsolescent Features".

Post-Compilation Rules

Certain pragmas are defined to be *configuration pragmas*; they shall appear before the first compilation_unit of a compilation. They are generally used to select a partition-wide or system-wide option. The pragma applies to all compilation_units appearing in the compilation, unless there are none, in which case it applies to all future compilation_units compiled into the same environment. 8

Implementation Permissions

An implementation may require that configuration pragmas that select partition-wide or system-wide options be compiled when the environment contains no library_items other than those of the predefined environment. In this case, the implementation shall still accept configuration pragmas in individual compilations that confirm the initially selected partition-wide or system-wide options. 9/2

Paragraph 10 was moved to Annex J, "Obsolescent Features".

10.1.6 Environment-Level Visibility Rules

1 The normal visibility rules do not apply within a parent_unit_name or a context_clause, nor within a pragma that appears at the place of a compilation unit. The special visibility rules for those contexts are given here.

Static Semantics

2/2 Within the parent_unit_name at the beginning of an explicit library_item, and within a nonlimited_with_clause, the only declarations that are visible are those that are explicit library_items of the environment, and the only declarations that are directly visible are those that are explicit root library_items of the environment. Within a limited_with_clause, the only declarations that are visible are those that are the implicit declaration of the limited view of a library package of the environment, and the only declarations that are directly visible are those that are the implicit declaration of the limited view of a root library package.

3 Within a use_clause or pragma that is within a context_clause, each library_item mentioned in a previous with_clause of the same context_clause is visible, and each root library_item so mentioned is directly visible. In addition, within such a use_clause, if a given declaration is visible or directly visible, each declaration that occurs immediately within the given declaration's visible part is also visible. No other declarations are visible or directly visible.

4 Within the parent_unit_name of a subunit, library_items are visible as they are in the parent_unit_name of a library_item; in addition, the declaration corresponding to each body_stub in the environment is also visible.

5 Within a pragma that appears at the place of a compilation unit, the immediately preceding library_item and each of its ancestors is visible. The ancestor root library_item is directly visible.

6/2 Notwithstanding the rules of 4.1.3, an expanded name in a with_clause, a pragma in a context_clause, or a pragma that appears at the place of a compilation unit may consist of a prefix that denotes a generic package and a selector_name that denotes a child of that generic package. (The child is necessarily a generic unit; see 10.1.1.)

10.2 Program Execution

1 An Ada *program* consists of a set of *partitions*, which can execute in parallel with one another, possibly in a separate address space, and possibly on a separate computer.

Post-Compilation Rules

2/5 A partition is a program or part of a program that can be invoked from outside the Ada implementation. For example, on many systems, a partition can be an executable file generated by the system linker. The user can *explicitly assign* library units to a partition. The assignment is done in an implementation-defined manner. The compilation units included in a partition are those of the explicitly assigned library units, as well as other compilation units *needed by* those library units. The compilation units needed by a given compilation unit (the *needed compilation units*) are determined as follows (unless specified otherwise via an implementation-defined pragma, or by some other implementation-defined means):

3/5 • A compilation unit is a needed compilation unit of itself;

4/5 • If a compilation unit is among the needed compilation units, then so are any compilation units upon which it depends semantically;

- If a library_unit_declaration is among the needed compilation units, then so is any corresponding library_unit_body; 5/5
- If a compilation unit with stubs is among the needed compilation units, then so are any corresponding subunits; 6/5
- If the (implicit) declaration of the limited view of a library package is among the needed compilation units, then so is the explicit declaration of the library package. 6.1/5

The user can optionally designate (in an implementation-defined manner) one subprogram as the *main subprogram* for the partition. A main subprogram, if specified, shall be a subprogram. 7

Each partition has an anonymous *environment task*, which is an implicit outermost task whose execution elaborates the library_items of the environment declarative_part, and then calls the main subprogram, if there is one. A partition's execution is that of its tasks. 8

The order of elaboration of library units is determined primarily by the *elaboration dependences*. There is an elaboration dependence of a given library_item upon another if the given library_item or any of its subunits depends semantically on the other library_item. In addition, if a given library_item or any of its subunits has a pragma Elaborate or Elaborate_All that names another library unit, then there is an elaboration dependence of the given library_item upon the body of the other library unit, and, for Elaborate_All only, upon each library_item that is a needed compilation unit of the declaration of the other library unit. 9/5

The environment task for a partition has the following structure: 10

```
task Environment_Task;                                                    11
task body Environment_Task is                                             12
   ... (1) -- The environment declarative_part
           -- (that is, the sequence of library_items) goes here.
begin
   ... (2) -- Call the main subprogram, if there is one.
end Environment_Task;
```

The environment declarative_part at (1) is a sequence of declarative_items consisting of copies of the library_items included in the partition. The order of elaboration of library_items is the order in which they appear in the environment declarative_part: 13

- The order of all included library_items is such that there are no forward elaboration dependences. 14
- Any included library_unit_declaration for which aspect Elaborate_Body is True is immediately followed by its library_unit_body, if included. 15/5
- All library_items declared pure occur before any that are not declared pure. 16
- All preelaborated library_items occur before any that are not preelaborated. 17

There shall be a total order of the library_items that obeys the above rules. The order is otherwise implementation defined. 18

The full expanded names of the library units and subunits included in a given partition shall be distinct. 19

The sequence_of_statements of the environment task (see (2) above) consists of either: 20

- A call to the main subprogram, if the partition has one. If the main subprogram has parameters, they are passed; where the actuals come from is implementation defined. What happens to the result of a main function is also implementation defined. 21

22 or:

23 • A null_statement, if there is no main subprogram.

24/5 The mechanisms for building and running partitions are implementation defined. These can be combined into one operation, as, for example, in dynamic linking, or “load-and-go” systems.

Dynamic Semantics

25 The execution of a program consists of the execution of a set of partitions. Further details are implementation defined. The execution of a partition starts with the execution of its environment task, ends when the environment task terminates, and includes the executions of all tasks of the partition. The execution of the (implicit) task_body of the environment task acts as a master for all other tasks created as part of the execution of the partition. When the environment task completes (normally or abnormally), it waits for the termination of all such tasks, and then finalizes any remaining objects of the partition.

Bounded (Run-Time) Errors

26 Once the environment task has awaited the termination of all other tasks of the partition, any further attempt to create a task (during finalization) is a bounded error, and may result in the raising of Program_Error either upon creation or activation of the task. If such a task is activated, it is not specified whether the task is awaited prior to termination of the environment task.

Implementation Requirements

27 The implementation shall ensure that all compilation units included in a partition are consistent with one another, and are legal according to the rules of the language.

Implementation Permissions

28/3 The kind of partition described in this subclause is known as an *active* partition. An implementation is allowed to support other kinds of partitions, with implementation-defined semantics.

29 An implementation may restrict the kinds of subprograms it supports as main subprograms. However, an implementation is required to support all main subprograms that are public parameterless library procedures.

30 If the environment task completes abnormally, the implementation may abort any dependent tasks.

31/5 NOTE 1 An implementation can provide inter-partition communication mechanism(s) via special packages and pragmas. Standard pragmas for distribution and methods for specifying inter-partition communication are defined in Annex E, “Distributed Systems”. If no such mechanisms are provided, then each partition is isolated from all others, and behaves as a program in and of itself.

32/5 NOTE 2 Partitions are not required to run in separate address spaces. For example, an implementation can support dynamic linking via the partition concept.

33/5 NOTE 3 An order of elaboration of library_items that is consistent with the partial ordering defined above does not always ensure that each library_unit_body is elaborated before any other compilation unit whose elaboration necessitates that the library_unit_body be already elaborated. (In particular, there is no requirement that the body of a library unit be elaborated as soon as possible after the library_unit_declaration is elaborated, unless the pragmas or aspects in 10.2.1 are used.)

34/5 NOTE 4 A partition (active or otherwise) does not necessarily have a main subprogram. In such a case, all the work done by the partition would be done by elaboration of various library_items, and by tasks created by that elaboration. Passive partitions, which cannot have main subprograms, are defined in Annex E, “Distributed Systems”.

10.2.1 Elaboration Control

This subclause defines aspects and pragmas that help control the elaboration order of library_items. 1/5

Paragraphs 2 through 4 were moved to Annex J, "Obsolescent Features".

Legality Rules

An elaborable construct is preelaborable unless its elaboration performs any of the following actions: 5

- The execution of a statement other than a null_statement. 6
- A call to a subprogram other than: 7/5
  - a static function; 7.1/5
  - an instance of Unchecked_Conversion (see 13.9); 7.2/5
  - a function declared in System.Storage_Elements (see 13.7.1); or 7.3/5
  - the functions To_Pointer and To_Address declared in an instance of System.Address_To_Access_Conversions (see 13.7.2). 7.4/5
- The evaluation of a primary that is a name of an object, unless the name is a static expression, or statically denotes a discriminant of an enclosing type. 8
- The creation of an object (including a component) that is initialized by default, if its type does not have preelaborable initialization. Similarly, the evaluation of an extension_aggregate with an ancestor subtype_mark denoting a subtype of such a type. 9/3
- The elaboration of any elaborable construct that is not preelaborable. 9.1/5

The actions of checks associated with Default_Initial_Condition, Type_Invariant, and Static_Predicate aspects are not considered when determining if a construct is preelaborable. 9.2/6

In determining whether the elaboration of an elaborable construct performs some action, an assertion policy of Check is assumed to be in effect everywhere and for all assertion aspects. Assertion_Policy pragmas are ignored in this determination. 9.3/6

A generic declaration is preelaborable unless every instance would perform one of the above actions. 9.4/6

A generic body is preelaborable only if elaboration of a corresponding instance body would not perform any such actions, presuming that: 10/2

- the actual for each discriminated formal derived type, formal private type, or formal private extension declared within the formal part of the generic unit is a type that does not have preelaborable initialization, unless the Preelaborable_Initialization aspect was specified for the formal type; 10.1/5
- the actual for each formal type is nonstatic; 10.2/2
- the actual for each formal object is nonstatic; and 10.3/2
- the actual for each formal subprogram is a user-defined subprogram. 10.4/2

When the library unit aspect (see 13.1.1) Preelaborate of a program unit is True, the unit is said to be *preelaborated*. When the Preelaborate aspect is specified True for a library unit, all compilation units of the library unit are preelaborated. The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable. All compilation units of a preelaborated library unit shall depend semantically only on declared pure or preelaborated library_items. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the 11/5

private part of an instance of a generic unit. If a library unit is preelaborated, then its declaration, if any, and body, if any, are elaborated prior to all nonpreelaborated library_items of the partition.

11.1/5 The following rules specify which entities have *preelaborable initialization*, namely that the Preelaborable_Initialization aspect of the entity is True:

11.2/5 - The partial view of a private type or private extension, a protected type without entry_declarations, a generic formal private type, or a generic formal derived type, has preelaborable initialization if and only if the Preelaborable_Initialization aspect has been specified True for them. A protected type with entry_declarations or a task type never has preelaborable initialization. The Preelaborable_Initialization aspect of a partial view of a type may be specified as False, even if the full view of the type has preelaborable initialization. Similarly, a generic formal type may be specified with Preelaborable_Initialization False, even if the actual type in an instance has preelaborable initialization.

11.3/6 - A component (including a discriminant) of a record or protected type has preelaborable initialization if no Dynamic_Predicate specification applies to its subtype, and:

11.4/6 - its declaration includes a default_expression whose execution does not perform any actions prohibited in preelaborable constructs as described above, or

11.5/6 - its declaration does not include a default_expression and its type has preelaborable initialization.

11.6/6 - A derived type has preelaborable initialization if its parent type has preelaborable initialization and if the noninherited components all have preelaborable initialization. However, a controlled type with an Initialize procedure that is not a null procedure does not have preelaborable initialization.

11.7/6 - An interface type has preelaborable initialization if and only if no Type_Invariant'Class aspects apply to it.

11.8/6 - A view of a type has preelaborable initialization if it is an elementary type, an array type whose component type has preelaborable initialization, or a record type whose components all have preelaborable initialization.

11.9/6 The following attribute is defined for a nonformal composite subtype S declared within the visible part of a package or a generic package, or a generic formal private subtype or formal derived subtype:

11.10/6 S'Preelaborable_Initialization

This attribute is of Boolean type, and its value reflects whether the type of S has preelaborable initialization. The value of this attribute, the type-related Preelaborable_Initialization aspect, may be specified for any type for which the attribute is defined. The value shall be specified by a static expression, unless the type is not a formal type but is nevertheless declared within a generic package. In this latter case, the value may also be specified by references to the Preelaborable_Initialization attribute of one or more formal types visible at the point of the declaration of the composite type, conjoined with **and**.

11.11/6 A function is *callable from preelaborable initialization* of type *T* if:

11.12/6 - It is a predefined operator;

11.13/6 - It is an expression function and all of the functions that occur in the return expression of the expression function are callable from preelaborable initialization of *T*;

11.14/6 - It is completed by an expression function in the same package specification as the type *T*, and all of the functions that occur in the return expression of the expression function are callable from preelaborable initialization of *T*;

- It is, or is declared in, a library unit which is named in a pragma Elaborate_All which applies to the compilation unit declaring *T*. 1.15/6

If the Preelaborable_Initialization aspect is specified True for a private type or a private extension *T*, then: 1.16/6

- the full view of the type shall have preelaborable initialization; 1.17/6
- no Type_Invariant'Class or Dynamic_Predicate aspect shall apply to the type; 1.18/6
- the expression of any Type_Invariant or Default_Initial_Condition aspect that applies shall not contain any function calls that are not callable from preelaborable initialization of type *T*. 1.19/6

If the aspect is specified True for a protected type, the protected type shall not have entries, and each component of the protected type shall have preelaborable initialization. If the aspect is specified True for a generic formal type, then in a generic_instantiation the corresponding actual type shall have preelaborable initialization. If the aspect definition includes one or more Preelaborable_Initialization attribute_references, then the full view of the type shall have preelaborable initialization presuming the types mentioned in the prefixes of the attribute_references all have preelaborable initialization. For any other composite type, the aspect shall be specified statically True or False only if it is confirming. In addition to the places where Legality Rules normally apply (see 12.3), these rules apply also in the private part of an instance of a generic unit. 1.20/6

For a type *T* that has Preelaborable_Initialization specified True, an object_declaration of type *T* shall not appear before the completion of any function that can be called by the evaluation of the expression of any Type_Invariant or Default_Initial_Condition that applies to *T*. 1.21/6

Implementation Advice

In an implementation, a type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package, whether the elaborations occur in distinct executions of the same program, or in executions of distinct programs or partitions that include the given version. 12

Paragraphs 13 through 15 were moved to Annex J, "Obsolescent Features".

Static Semantics

A *pure* program unit is a preelaborable program unit whose elaboration does not perform any of the following actions: 15.1/5

- the elaboration of a variable declaration; 15.2/2
- the evaluation of an allocator of an access-to-variable type; for the purposes of this rule, the partial view of a type is presumed to have nonvisible components whose default initialization evaluates such an allocator; 15.3/2
- the elaboration of the declaration of a nonderived named access-to-variable type unless the Storage_Size of the type has been specified by a static expression with value zero or is defined by the language to be zero; 15.4/3
- the elaboration of the declaration of a nonderived named access-to-constant type for which the Storage_Size has been specified by an expression other than a static expression with value zero; 15.5/5
- the elaboration of any program unit that is not pure. 15.6/5

A generic declaration is pure unless every instance would perform one of the above actions. 15.7/5

A generic body is pure only if elaboration of a corresponding instance body would not perform any such actions presuming any composite formal types have nonvisible components whose default initialization evaluates an allocator of an access-to-variable type. 15.8/5

15.9/5 The Storage_Size for an anonymous access-to-variable type declared at library level in a library unit that is declared pure is defined to be zero.

Legality Rules

16/2 *This paragraph was deleted.*

17/5 When the library unit aspect Pure of a program unit is True, the unit is said to be *declared pure*. When the Pure aspect is specified True for a library unit, all compilation units of the library unit are declared pure. In addition, the limited view of any library package is declared pure. The declaration and body of a declared pure library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be pure. All compilation units of a declared pure library unit shall depend semantically only on declared pure library_items. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. Furthermore, the full view of any partial view declared in the visible part of a declared pure library unit that has any available stream attributes shall support external streaming (see 13.13.2).

Erroneous Execution

17.1/4 Execution is erroneous if some operation (other than the initialization or finalization of the object) modifies the value of a constant object declared at library-level in a pure package.

Implementation Permissions

18/6 If a library unit is declared pure, then the implementation is permitted to omit a call on a library-level subprogram of the library unit if the results are not needed after the call. In addition, the implementation may omit a call on such a subprogram and simply reuse the results produced by an earlier call on the same subprogram, provided that:

18.1/6 - none of the parameters nor any object accessible via access values from the parameters have any part that is of a type whose full type is an immutably limited type; and

18.2/6 - the addresses and values of all by-reference actual parameters, the values of all by-copy-in actual parameters, and the values of all objects accessible via access values from the parameters, are the same as they were at the earlier call.

18.3/6 This permission applies even if the subprogram produces other side effects when called.

Syntax

19/5 The following pragmas are defined with the given forms:

20 **pragma** Elaborate(*library_unit*_name{, *library_unit*_name});

21 **pragma** Elaborate_All(*library_unit*_name{, *library_unit*_name});

22/5 *This paragraph was deleted.*

23/6 A pragma Elaborate or Elaborate_All is a context pragma (see 2.8), and is only allowed within a context_clause.

24/5 *This paragraph was deleted.*

Legality Rules

25/6 If the library unit aspect Elaborate_Body is True for a declaration, then the declaration requires a completion (a body).

The *library_unit_*name of a pragma Elaborate or Elaborate_All shall denote a nonlimited view of a library unit. 25.1/2

Static Semantics

A pragma Elaborate specifies that the body of the named library unit is elaborated before the current library_item. A pragma Elaborate_All specifies that each library_item that is needed by the named library unit declaration is elaborated before the current library_item. 26/3

If the Elaborate_Body aspect of a library unit is True, the body of the library unit is elaborated immediately after its declaration. 26.1/5

NOTE 1 A preelaborated library unit can have nonpreelaborable children. 27/5

NOTE 2 A library unit that is declared pure can have impure children. 28/5

11 Exceptions

This clause defines the facilities for dealing with errors or other exceptional situations that arise during program execution. An *exception* represents a kind of exceptional situation; an occurrence of such a situation (at run time) is called an *exception occurrence*. To *raise* an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. Performing some actions in response to the arising of an exception is called *handling* the exception. 1/3

An exception_declaration declares a name for an exception. An exception can be raised explicitly (for example, by a raise_statement) or implicitly (for example, by the failure of a language-defined check). When an exception arises, control can be transferred to a user-provided exception_handler at the end of a handled_sequence_of_statements, or it can be propagated to a dynamically enclosing execution. 2/3

11.1 Exception Declarations

An exception_declaration declares a name for an exception. 1

Syntax

exception_declaration ::= defining_identifier_list : **exception** [aspect_specification]; 2/3

Static Semantics

Each single exception_declaration declares a name for a different exception. If a generic unit includes an exception_declaration, the exception_declarations implicitly generated by different instantiations of the generic unit refer to distinct exceptions (but all have the same defining_identifier). The particular exception denoted by an exception name is determined at compilation time and is the same regardless of how many times the exception_declaration is elaborated. 3

The *predefined* exceptions are the ones declared in the declaration of package Standard: Constraint_Error, Program_Error, Storage_Error, and Tasking_Error; one of them is raised when a language-defined check fails. 4

Dynamic Semantics

The elaboration of an exception_declaration has no effect. 5

The execution of any construct raises Storage_Error if there is insufficient storage for that execution. The amount of storage necessary for the execution of constructs is unspecified. 6/5

Examples

Examples of user-defined exception declarations: 7

```
Singular : exception;                                    8
Error    : exception;
Overflow, Underflow : exception;
```

11.2 Exception Handlers

1 The response to one or more exceptions is specified by an exception_handler.

Syntax

2
```
handled_sequence_of_statements ::=
   sequence_of_statements
  [exception
   exception_handler
   {exception_handler}]
```

3/5
```
exception_handler ::=
  when [choice_parameter_specification:] exception_choice {'|' exception_choice} =>
   sequence_of_statements
```

4
```
choice_parameter_specification ::= defining_identifier
```

5
```
exception_choice ::= exception_name | others
```

Legality Rules

5.1/4 An *exception_*name of an exception_choice shall denote an exception.

6 A choice with an *exception_*name *covers* the named exception. A choice with **others** covers all exceptions not named by previous choices of the same handled_sequence_of_statements. Two choices in different exception_handlers of the same handled_sequence_of_statements shall not cover the same exception.

7 A choice with **others** is allowed only for the last handler of a handled_sequence_of_statements and as the only choice of that handler.

8 An *exception_*name of a choice shall not denote an exception declared in a generic formal package.

Static Semantics

9 A choice_parameter_specification declares a *choice parameter*, which is a constant object of type Exception_Occurrence (see 11.4.1). During the handling of an exception occurrence, the choice parameter, if any, of the handler represents the exception occurrence that is being handled.

Dynamic Semantics

10 The execution of a handled_sequence_of_statements consists of the execution of the sequence_of_statements. The optional handlers are used to handle any exceptions that are propagated by the sequence_of_statements.

Examples

11 *Example of an exception handler:*

12/5
```
begin
   Open(File, In_File, "input.txt");    -- see A.8.2
exception
   when E : Name_Error =>
      Put("Cannot open input file : ");
      Put_Line(Ada.Exceptions.Exception_Message(E));  -- see 11.4.1
      raise;
end;
```

11.3 Raise Statements and Raise Expressions

A raise_statement raises an exception. 1

Syntax

raise_statement ::= **raise**; 2/2
| **raise** *exception*_name [**with** *string*_expression];

raise_expression ::= **raise** *exception*_name [**with** *string*_simple_expression] 2.1/4

If a raise_expression appears within the expression of one of the following contexts, the raise_expression shall appear within a pair of parentheses within the expression: 2.2/4

- object_declaration; 2.3/4
- modular_type_definition; 2.4/4
- floating_point_definition; 2.5/4
- ordinary_fixed_point_definition; 2.6/4
- decimal_fixed_point_definition; 2.7/4
- default_expression; 2.8/4
- ancestor_part. 2.9/4

Legality Rules

The *exception*_name, if any, of a raise_statement or raise_expression shall denote an exception. A raise_statement with no *exception*_name (that is, a *re-raise statement*) shall be within a handler, but not within a body enclosed by that handler. 3/4

Name Resolution Rules

The *string*_expression or *string*_simple_expression, if any, of a raise_statement or raise_expression is expected to be of type String. 3.1/4

The expected type for a raise_expression shall be any single type. 3.2/4

Dynamic Semantics

To *raise an exception* is to raise a new occurrence of that exception, as explained in 11.4. For the execution of a raise_statement with an *exception*_name, the named exception is raised. Similarly, for the evaluation of a raise_expression, the named exception is raised. In both of these cases, if a *string*_expression or *string*_simple_expression is present, the expression is evaluated and its value is associated with the exception occurrence. For the execution of a re-raise statement, the exception occurrence that caused transfer of control to the innermost enclosing handler is raised again. 4/4

NOTE If the evaluation of a *string*_expression or *string*_simple_expression raises an exception, that exception is propagated instead of the one denoted by the *exception*_name of the raise_statement or raise_expression. 4.1/4

Examples

Examples of raise statements: 5

```
raise Ada.IO_Exceptions.Name_Error;    -- see A.13
raise Queue_Error with "Buffer Full"; -- see 9.11
```
6/2

```
raise;                                 -- re-raise the current exception
```
7

-- For an example of a raise expression, see the Streams Subsystem definitions in 13.13.1. 8/5

11.4 Exception Handling

1 When an exception occurrence is raised, normal program execution is abandoned and control is transferred to an applicable exception_handler, if any. To *handle* an exception occurrence is to respond to the exceptional event. To *propagate* an exception occurrence is to raise it again in another context; that is, to fail to respond to the exceptional event in the present context.

Dynamic Semantics

2 Within a given task, if the execution of construct *a* is defined by this document to consist (in part) of the execution of construct *b*, then while *b* is executing, the execution of *a* is said to *dynamically enclose* the execution of *b*. The *innermost dynamically enclosing* execution of a given execution is the dynamically enclosing execution that started most recently.

3 When an exception occurrence is raised by the execution of a given construct, the rest of the execution of that construct is *abandoned*; that is, any portions of the execution that have not yet taken place are not performed. The construct is first completed, and then left, as explained in 7.6.1. Then:

4 • If the construct is a task_body, the exception does not propagate further;

5 • If the construct is the sequence_of_statements of a handled_sequence_of_statements that has a handler with a choice covering the exception, the occurrence is handled by that handler;

6 • Otherwise, the occurrence is *propagated* to the innermost dynamically enclosing execution, which means that the occurrence is raised again in that context.

7 When an occurrence is *handled* by a given handler, the choice_parameter_specification, if any, is first elaborated, which creates the choice parameter and initializes it to the occurrence. Then, the sequence_of_statements of the handler is executed; this execution replaces the abandoned portion of the execution of the sequence_of_statements.

8/5 NOTE Exceptions raised in a declarative_part of a body are not handled by the handlers of the handled_sequence_of_-statements of that body.

11.4.1 The Package Exceptions

Static Semantics

1 The following language-defined library package exists:

2/5
```
with Ada.Streams;
package Ada.Exceptions
    with Preelaborate, Nonblocking, Global => in out synchronized is
    type Exception_Id is private
        with Preelaborable_Initialization;
    Null_Id : constant Exception_Id;
    function Exception_Name(Id : Exception_Id) return String;
    function Wide_Exception_Name(Id : Exception_Id) return Wide_String;
    function Wide_Wide_Exception_Name(Id : Exception_Id)
        return Wide_Wide_String;
```

3/5
```
    type Exception_Occurrence is limited private
        with Preelaborable_Initialization;
    type Exception_Occurrence_Access is access all Exception_Occurrence;
    Null_Occurrence : constant Exception_Occurrence;
```

```
   procedure Raise_Exception(E : in Exception_Id;                                    4/3
                             Message : in String := "")
       with No_Return;
   function Exception_Message(X : Exception_Occurrence) return String;
   procedure Reraise_Occurrence(X : in Exception_Occurrence);

   function Exception_Identity(X : Exception_Occurrence)                             5/2
                               return Exception_Id;
   function Exception_Name(X : Exception_Occurrence) return String;
       -- Same as Exception_Name(Exception_Identity(X)).
   function Wide_Exception_Name(X : Exception_Occurrence)
       return Wide_String;
       -- Same as Wide_Exception_Name(Exception_Identity(X)).
   function Wide_Wide_Exception_Name(X : Exception_Occurrence)
       return Wide_Wide_String;
       -- Same as Wide_Wide_Exception_Name(Exception_Identity(X)).
   function Exception_Information(X : Exception_Occurrence) return String;

   procedure Save_Occurrence(Target : out Exception_Occurrence;                      6/2
                             Source : in Exception_Occurrence);
   function Save_Occurrence(Source : Exception_Occurrence)
                            return Exception_Occurrence_Access;

   procedure Read_Exception_Occurrence                                               6.1/2
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : out Exception_Occurrence);
   procedure Write_Exception_Occurrence
      (Stream : not null access Ada.Streams.Root_Stream_Type'Class;
       Item   : in Exception_Occurrence);

   for Exception_Occurrence'Read use Read_Exception_Occurrence;                      6.2/2
   for Exception_Occurrence'Write use Write_Exception_Occurrence;

private                                                                              6.3/2
   ... -- not specified by the language
end Ada.Exceptions;
```

Each distinct exception is represented by a distinct value of type Exception_Id. Null_Id does not represent any exception, and is the default initial value of type Exception_Id. Each occurrence of an exception is represented by a value of type Exception_Occurrence. Null_Occurrence does not represent any exception occurrence, and is the default initial value of type Exception_Occurrence. 7

For a prefix E that denotes an exception, the following attribute is defined: 8/1

E'Identity E'Identity returns the unique identity of the exception. The type of this attribute is Exception_Id. 9

Raise_Exception raises a new occurrence of the identified exception. 10/2

Exception_Message returns the message associated with the given Exception_Occurrence. For an occurrence raised by a call to Raise_Exception, the message is the Message parameter passed to Raise_Exception. For the occurrence raised by a raise_statement or raise_expression with an *exception_*name and a *string_*expression or *string_*simple_expression, the message is the *string_*expression or *string_*simple_expression. For the occurrence raised by a raise_statement or raise_expression with an *exception_*name but without a *string_*expression or *string_*simple_expression, the message is a string giving implementation-defined information about the exception occurrence. For an occurrence originally raised in some other manner (including by the failure of a language-defined check), the message is an unspecified string. In all cases, Exception_Message returns a string with lower bound 1. 10.1/4

Reraise_Occurrence reraises the specified exception occurrence. 10.2/2

Exception_Identity returns the identity of the exception of the occurrence. 11

The Wide_Wide_Exception_Name functions return the full expanded name of the exception, in upper case, starting with a root library unit. For an exception declared immediately within package Standard, the 12/2

defining_identifier is returned. The result is implementation defined if the exception is declared within an unnamed block_statement.

12.1/2 The Exception_Name functions (respectively, Wide_Exception_Name) return the same sequence of graphic characters as that defined for Wide_Wide_Exception_Name, if all the graphic characters are defined in Character (respectively, Wide_Character); otherwise, the sequence of characters is implementation defined, but no shorter than that returned by Wide_Wide_Exception_Name for the same value of the argument.

12.2/2 The string returned by the Exception_Name, Wide_Exception_Name, and Wide_Wide_Exception_Name functions has lower bound 1.

13/2 Exception_Information returns implementation-defined information about the exception occurrence. The returned string has lower bound 1.

14/2 Reraise_Occurrence has no effect in the case of Null_Occurrence. Raise_Exception and Exception_Name raise Constraint_Error for a Null_Id. Exception_Message, Exception_Name, and Exception_Information raise Constraint_Error for a Null_Occurrence. Exception_Identity applied to Null_Occurrence returns Null_Id.

15 The Save_Occurrence procedure copies the Source to the Target. The Save_Occurrence function uses an allocator of type Exception_Occurrence_Access to create a new object, copies the Source to this new object, and returns an access value designating this new object; the result may be deallocated using an instance of Unchecked_Deallocation.

15.1/2 Write_Exception_Occurrence writes a representation of an exception occurrence to a stream; Read_Exception_Occurrence reconstructs an exception occurrence from a stream (including one written in a different partition).

Paragraph 16 was deleted.

Implementation Permissions

17 An implementation of Exception_Name in a space-constrained environment may return the defining_identifier instead of the full expanded name.

18 The string returned by Exception_Message may be truncated (to no less than 200 characters) by the Save_Occurrence procedure (not the function), the Reraise_Occurrence procedure, and the re-raise statement.

Implementation Advice

19 Exception_Message (by default) and Exception_Information should produce information useful for debugging. Exception_Message should be short (about one line), whereas Exception_Information can be long. Exception_Message should not include the Exception_Name. Exception_Information should include both the Exception_Name and the Exception_Message.

20/5 NOTE UTF-8 encoding (see A.4.11) can be used to represent non-ASCII characters in exception messages.

11.4.2 Pragmas Assert and Assertion_Policy

Pragma Assert is used to assert the truth of a boolean expression at a point within a sequence of declarations or statements. 1/3

Assert pragmas, subtype predicates (see 3.2.4), preconditions and postconditions (see 6.1.1), type invariants (see 7.3.2), and default initial conditions (see 7.3.3) are collectively referred to as *assertions*; their boolean expressions are referred to as *assertion expressions*. 1.1/5

Pragma Assertion_Policy is used to control whether assertions are to be ignored by the implementation, checked at run time, or handled in some implementation-defined manner. 1.2/3

Syntax

The form of a pragma Assert is as follows: 2/2

pragma Assert([Check =>] *boolean_*expression[, [Message =>] *string_*expression]); 3/2

A pragma Assert is an executable pragma (see 2.8), and is allowed at the place where a basic_declaration, declare_item, or statement is allowed. 4/6

The form of a pragma Assertion_Policy is as follows: 5/2

pragma Assertion_Policy(*policy_*identifier); 6/2

pragma Assertion_Policy(6.1/3
*assertion_*aspect_mark => *policy_*identifier
{, *assertion_*aspect_mark => *policy_*identifier});

A pragma Assertion_Policy is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma. 7/3

Name Resolution Rules

The expected type for the *boolean_*expression of a pragma Assert is any boolean type. The expected type for the *string_*expression of a pragma Assert is type String. 8/2

Legality Rules

The *assertion_*aspect_mark of a pragma Assertion_Policy shall identify an *assertion aspect*, namely one of Assert, Static_Predicate, Dynamic_Predicate, Pre, Pre'Class, Post, Post'Class, Type_Invariant, Type_Invariant'Class, Default_Initial_Condition, or some implementation-defined (assertion) aspect_mark. The *policy_*identifier shall be either Check, Ignore, or some implementation-defined identifier. 9/5

Static Semantics

A pragma Assertion_Policy determines for each assertion aspect named in the pragma_argument_associations whether assertions of the given aspect are to be enforced by a runtime check. The *policy_*identifier Check requires that assertion expressions of the given aspect be checked that they evaluate to True at the points specified for the given aspect; the *policy_*identifier Ignore requires that the assertion expression not be evaluated at these points, and the runtime checks not be performed. Note that for subtype predicate aspects (see 3.2.4), even when the applicable Assertion_Policy is Ignore, the predicate will still be evaluated as part of membership tests and Valid attribute_references, and if static, will still have an effect on loop iteration over the subtype, and the selection of case_statement_alternatives and variants. 10/3

10.1/3 If no *assertion*_aspect_marks are specified in the pragma, the specified policy applies to all assertion aspects.

10.2/5 A pragma Assertion_Policy applies to the named assertion aspects in a specific region, and applies to all assertion expressions associated with those aspects specified in that region. A pragma Assertion_Policy given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a pragma Assertion_Policy given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies.

10.3/3 If a pragma Assertion_Policy applies to a generic_instantiation, then the pragma Assertion_Policy applies to the entire instance.

10.4/3 If multiple Assertion_Policy pragmas apply to a given construct for a given assertion aspect, the assertion policy is determined by the one in the innermost enclosing region of a pragma Assertion_Policy specifying a policy for the assertion aspect. If no such Assertion_Policy pragma exists, the policy is implementation defined.

11/2 The following language-defined library package exists:

12/5
```
package Ada.Assertions
    with Pure is
```

13/2
```
   Assertion_Error : exception;
```

14/2
```
   procedure Assert(Check : in Boolean);
   procedure Assert(Check : in Boolean; Message : in String);
```

15/2
```
end Ada.Assertions;
```

16/3 A compilation unit containing a check for an assertion (including a pragma Assert) has a semantic dependence on the Assertions library unit.

17/3 *This paragraph was deleted.*

Dynamic Semantics

18/3 If performing checks is required by the Assert assertion policy in effect at the place of a pragma Assert, the elaboration of the pragma consists of evaluating the boolean expression, and if the result is False, evaluating the Message argument, if any, and raising the exception Assertions.Assertion_Error, with a message if the Message argument is provided.

19/2 Calling the procedure Assertions.Assert without a Message parameter is equivalent to:

20/2
```
if Check = False then
   raise Ada.Assertions.Assertion_Error;
end if;
```

21/2 Calling the procedure Assertions.Assert with a Message parameter is equivalent to:

22/2
```
if Check = False then
   raise Ada.Assertions.Assertion_Error with Message;
end if;
```

23/2 The procedures Assertions.Assert have these effects independently of the assertion policy in effect.

23.1/6 If two or more expressions are required to be fully conformant then only the assertion policy in effect at the place of the first of the expressions is considered in determining the dynamic semantics of evaluating the expression.

Bounded (Run-Time) Errors

It is a bounded error to invoke a potentially blocking operation (see 9.5.1) during the evaluation of an assertion expression associated with a call on, or return from, a protected operation. If the bounded error is detected, Program_Error is raised. If not detected, execution proceeds normally, but if it is invoked within a protected action, it can result in deadlock or a (nested) protected action. 23.2/6

Implementation Requirements

Any postcondition expression, type invariant expression, or default initial condition expression occurring in the specification of a language-defined unit is enabled (see 6.1.1, 7.3.2, and 7.3.3). 23.3/6

The evaluation of any such postcondition, type invariant, or default initial condition expression shall either yield True or propagate an exception from a raise_expression that appears within the assertion expression. 23.4/6

Any precondition expression occurring in the specification of a language-defined unit is enabled (see 6.1.1) unless suppressed (see 11.5). Similarly, any predicate checks for a subtype occurring in the specification of a language-defined unit are enabled (see 3.2.4) unless suppressed. 23.5/6

Implementation Permissions

Assertion_Error may be declared by renaming an implementation-defined exception from another package. 24/2

Implementations may define their own assertion policies. 25/2

If the result of a function call in an assertion is not used to determine the value of the assertion expression, an implementation is permitted to omit the function call. This permission applies even if the function has side effects. 26/5

An implementation may disallow the specification of an assertion expression if the evaluation of the expression has a side effect such that an immediate reevaluation of the expression can produce a different value. Similarly, an implementation may disallow the specification of an assertion expression that is checked as part of a call on or return from a callable entity *C*, if the evaluation of the expression has a side effect such that the evaluation of some other assertion expression associated with the same call of (or return from) *C* can produce a different value than in the case when the first expression had not been evaluated. 27/5

NOTE 1 Normally, the boolean expression in a pragma Assert should not call functions that have significant side effects when the result of the expression is True, so that the particular assertion policy in effect will not affect normal operation of the program. 28/5

NOTE 2 In most cases, adding or deleting a legal Assertion_Policy pragma has no impact on the legality or the static semantics of any construct. In particular, the rules pertaining to freezing (see 13.14) and the definition of preelaborability (see 10.2.1) are unaffected by Assertion_Policy pragmas. However, this general principle does not always hold in the case of a Restrictions pragma (see 13.12). 29/6

11.4.3 Example of Exception Handling

Examples

Exception handling can be used to separate the detection of an error from the response to that error: 1/5

```
package File_System is                                                    2/5
    type Data_Type is ...;
    type File_Handle is limited private;
```

```
3           File_Not_Found : exception;
            procedure Open(F : in out File_Handle; Name : String);
                -- raises File_Not_Found if named file does not exist
4           End_Of_File : exception;
            procedure Read(F : in out File_Handle; Data : out Data_Type);
                -- raises End_Of_File if the file is not open
5/5         ...
        private
            ...
        end File_System;
5.1/5   package body File_System is
            ...
6/5         procedure Open(F : in out File_Handle; Name : String) is
            begin
                if File_Exists(Name) then
                    ...
                else
                    raise File_Not_Found with "File not found: " & Name & ".";
                end if;
            end Open;
7           procedure Read(F : in out File_Handle; Data : out Data_Type) is
            begin
                if F.Current_Position <= F.Last_Position then
                    ...
                else
                    raise End_Of_File;
                end if;
            end Read;
8           ...
9       end File_System;
10/5    with Ada.Text_IO;
        with Ada.Exceptions;
        with File_System; use File_System;
        use Ada;
        procedure Main is
            Verbosity_Desired : Boolean := ...;
        begin
            ... -- call operations in File_System
        exception
            when End_Of_File =>
                Close(Some_File);
            when Not_Found_Error : File_Not_Found =>
                Text_IO.Put_Line(Exceptions.Exception_Message(Not_Found_Error));
            when The_Error : others =>
                Text_IO.Put_Line("Unknown error:");
                if Verbosity_Desired then
                    Text_IO.Put_Line(Exceptions.Exception_Information(The_Error));
                else
                    Text_IO.Put_Line(Exceptions.Exception_Name(The_Error));
                    Text_IO.Put_Line(Exceptions.Exception_Message(The_Error));
                end if;
                raise;
        end Main;
```

11/5 In the above example, the File_System package contains information about detecting certain exceptional situations, but it does not specify how to handle those situations. Procedure Main specifies how to handle them; other clients of File_System can have different handlers, even though the exceptional situations arise from the same basic causes.

11.5 Suppressing Checks

Checking pragmas give instructions to an implementation on handling language-defined checks. A pragma Suppress gives permission to an implementation to omit certain language-defined checks, while a pragma Unsuppress revokes the permission to omit checks. 1/2

A *language-defined check* (or simply, a "check") is one of the situations defined by this document that requires a check to be made at run time to determine whether some condition is true. A check *fails* when the condition being checked is False, causing an exception to be raised. 2/3

Syntax

The forms of checking pragmas are as follows: 3/2

pragma Suppress(identifier); 4/2

pragma Unsuppress(identifier); 4.1/2

A checking pragma is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma. 5/2

Legality Rules

The identifier shall be the name of a check. 6/2

This paragraph was deleted. 7/2

Static Semantics

A checking pragma applies to the named check in a specific region, and applies to all entities in that region. A checking pragma given in a declarative_part or immediately within a package_specification applies from the place of the pragma to the end of the innermost enclosing declarative region. The region for a checking pragma given as a configuration pragma is the declarative region for the entire compilation unit (or units) to which it applies. 7.1/2

If a checking pragma applies to a generic_instantiation, then the checking pragma also applies to the entire instance. 7.2/3

A pragma Suppress gives permission to an implementation to omit the named check (or every check in the case of All_Checks) for any entities to which it applies. If permission has been given to suppress a given check, the check is said to be *suppressed*. 8/2

A pragma Unsuppress revokes the permission to omit the named check (or every check in the case of All_Checks) given by any pragma Suppress that applies at the point of the pragma Unsuppress. The permission is revoked for the region to which the pragma Unsuppress applies. If there is no such permission at the point of a pragma Unsuppress, then the pragma has no effect. A later pragma Suppress can renew the permission. 8.1/2

In the case of evaluating a default_expression or default_name, the applicable check suppression is determined by the place where the evaluation occurs, and not by where the construct occurs. 8.2/6

The following are the language-defined checks: 9

- The following checks correspond to situations in which the exception Constraint_Error is raised upon failure of a language-defined check. 10/5

11/2 Access_Check
When evaluating a dereference (explicit or implicit), check that the value of the name is not **null**. When converting to a subtype that excludes null, check that the converted value is not **null**.

12 Discriminant_Check
Check that the discriminants of a composite value have the values imposed by a discriminant constraint. Also, when accessing a record component, check that it exists for the current discriminant values.

13/2 Division_Check
Check that the second operand is not zero for the operations /, **rem** and **mod**.

14 Index_Check
Check that the bounds of an array value are equal to the corresponding bounds of an index constraint. Also, when accessing a component of an array object, check for each dimension that the given index value belongs to the range defined by the bounds of the array object. Also, when accessing a slice of an array object, check that the given discrete range is compatible with the range defined by the bounds of the array object.

15 Length_Check
Check that two arrays have matching components, in the case of array subtype conversions, and logical operators for arrays of boolean components.

16 Overflow_Check
Check that a scalar value is within the base range of its type, in cases where the implementation chooses to raise an exception instead of returning the correct mathematical result.

17/5 Range_Check
Check that a scalar value satisfies a range constraint. Also, for the elaboration of a subtype_indication, check that the constraint (if present) is compatible with the subtype denoted by the subtype_mark. Also, for an aggregate, check that an index or discriminant value belongs to the corresponding subtype. Also, check that when the result of an operation yields an array, the value of each component belongs to the component subtype. Also, for the attributes Value, Wide_Value, and Wide_Wide_Value, check that the given string has the appropriate syntax and value for the base subtype of the prefix of the attribute_reference.

18 Tag_Check
Check that operand tags in a dispatching call are all equal. Check for the correct tag on tagged type conversions, for an assignment_statement, and when returning a tagged limited object from a function.

19/5 - The following checks correspond to situations in which the exception Program_Error is raised upon failure of a language-defined check.

19.1/2 Accessibility_Check
Check the accessibility level of an entity or view.

19.2/2 Allocation_Check
For an allocator, check that the master of any tasks to be created by the allocator is not yet completed or some dependents have not yet terminated, and that the finalization of the collection has not started.

20 Elaboration_Check
When a subprogram or protected entry is called, a task activation is accomplished, or a generic instantiation is elaborated, check that the body of the corresponding unit has already been elaborated.

Program_Error_Check 21/5
Other language-defined checks that raise Program_Error: that subtypes with predicates are not used to index an array in a generic unit; that the maximum number of chunks is greater than zero; that the default value of an out parameter is convertible; that there is no misuse of functions in a generic with a class-wide actual type; that there are not colliding External_Tag values; that there is no misuse of operations of unchecked union types.

- The following check corresponds to situations in which the exception Storage_Error is raised upon failure of a language-defined check. 22/5

Storage_Check 23
Check that evaluation of an allocator does not require more space than is available for a storage pool. Check that the space available for a task or subprogram has not been exceeded.

- The following check corresponds to situations in which the exception Tasking_Error is raised upon failure of a language-defined check. 23.1/5

Tasking_Check 23.2/5
Check that all tasks activated successfully. Check that a called task has not yet terminated.

- The following checks correspond to situations in which the exception Assertion_Error is raised upon failure of a language-defined check. For a language-defined unit *U* associated with one of these checks in the list below, the check refers to performance of checks associated with the Pre, Static_Predicate, and Dynamic_Predicate aspects associated with any entity declared in a descendant of *U*, or in an instance of a generic unit which is, or is declared in, a descendant of *U*. Each check is associated with one or more units: 23.3/5

Calendar_Assertion_Check 23.4/5
Calendar.

Characters_Assertion_Check 23.5/5
Characters, Wide_Characters, and Wide_Wide_Characters.

Containers_Assertion_Check 23.6/5
Containers.

Interfaces_Assertion_Check 23.7/5
Interfaces.

IO_Assertion_Check 23.8/5
Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, Storage_IO, Streams.Stream_IO, and Directories.

Numerics_Assertion_Check 23.9/5
Numerics.

Strings_Assertion_Check 23.10/5
Strings.

System_Assertion_Check 23.11/5
System.

- The following check corresponds to all situations in which any predefined exception is raised upon failure of a check. 24/5

All_Checks 25/3
Represents the union of all checks; suppressing All_Checks suppresses all checks other than those associated with assertions. In addition, an implementation is allowed

(but not required) to behave as if a pragma Assertion_Policy(Ignore) applies to any region to which pragma Suppress(All_Checks) applies.

Erroneous Execution

26/5 If a given check has been suppressed, and the corresponding error situation occurs, the execution of the program is erroneous. Similarly, if a precondition check has been suppressed and the evaluation of the precondition would have raised an exception, execution is erroneous.

Implementation Permissions

27/2 An implementation is allowed to place restrictions on checking pragmas, subject only to the requirement that pragma Unsuppress shall allow any check names supported by pragma Suppress. An implementation is allowed to add additional check names, with implementation-defined semantics. When Overflow_Check has been suppressed, an implementation may also suppress an unspecified subset of the Range_Checks.

27.1/2 An implementation may support an additional parameter on pragma Unsuppress similar to the one allowed for pragma Suppress (see J.10). The meaning of such a parameter is implementation-defined.

Implementation Advice

28 The implementation should minimize the code executed for checks that have been suppressed.

29/5 NOTE 1 There is no guarantee that a suppressed check is actually removed; hence a pragma Suppress is useful only to improve efficiency.

29.1/2 NOTE 2 It is possible to give both a pragma Suppress and Unsuppress for the same check immediately within the same declarative_part. In that case, the last pragma given determines whether or not the check is suppressed. Similarly, it is possible to resuppress a check which has been unsuppressed by giving a pragma Suppress in an inner declarative region.

Examples

30/2 *Examples of suppressing and unsuppressing checks:*

31/2
```
pragma Suppress(Index_Check);
pragma Unsuppress(Overflow_Check);
```

11.6 Exceptions and Optimization

1/3 This subclause gives permission to the implementation to perform certain “optimizations” that do not necessarily preserve the canonical semantics.

Dynamic Semantics

2/3 The rest of this Reference Manual (outside this subclause) defines the *canonical semantics* of the language. The canonical semantics of a given (legal) program determines a set of possible external effects that can result from the execution of the program with given inputs.

3/3 As explained in 1.1.3, “Conformity of an Implementation”, the external effect of a program is defined in terms of its interactions with its external environment. Hence, the implementation can perform any internal actions whatsoever, in any order or in parallel, so long as the external effect of the execution of the program is one that is allowed by the canonical semantics, or by the rules of this subclause.

Implementation Permissions

4 The following additional permissions are granted to the implementation:

5/5 - An implementation can omit raising an exception when a language-defined check fails. Instead, the operation that failed the check can simply yield an *undefined result*. The exception is required to be raised by the implementation only if, in the absence of raising it, the value of this

undefined result would have some effect on the external interactions of the program. In determining this, the implementation shall not presume that an undefined result has a value that belongs to its subtype, nor even to the base range of its type, if scalar. Having removed the raise of the exception, the canonical semantics will in general allow the implementation to omit the code for the check, and some or all of the operation itself.

- If an exception is raised due to the failure of a language-defined check, then upon reaching the corresponding exception_handler (or the termination of the task, if none), the external interactions that have occurred have to reflect only that the exception was raised somewhere within the execution of the sequence_of_statements with the handler (or the task_body), possibly earlier (or later if the interactions are independent of the result of the checked operation) than that defined by the canonical semantics, but not within the execution of some abort-deferred operation or *independent* subprogram that does not dynamically enclose the execution of the construct whose check failed. An independent subprogram is one that is defined outside the library unit containing the construct whose check failed, and for which the Inline aspect is False. Any assignment that occurred outside of such abort-deferred operations or independent subprograms can be disrupted by the raising of the exception, causing the object or its parts to become abnormal, and certain subsequent uses of the object to be erroneous, as explained in 13.9.1. 6/5

NOTE The permissions granted by this subclause can have an effect on the semantics of a program only if the program fails a language-defined check. 7/3

12 Generic Units

A *generic unit* is a program unit that is either a generic subprogram or a generic package. A generic unit is a *template*, which can be parameterized, and from which corresponding (nongeneric) subprograms or packages can be obtained. The resulting program units are said to be *instances* of the original generic unit. 1

A generic unit is declared by a generic_declaration. This form of declaration has a generic_formal_part declaring any generic formal parameters. An instance of a generic unit is obtained as the result of a generic_instantiation with appropriate generic actual parameters for the generic formal parameters. An instance of a generic subprogram is a subprogram. An instance of a generic package is a package. 2

Generic units are templates. As templates they do not have the properties that are specific to their nongeneric counterparts. For example, a generic subprogram can be instantiated but it cannot be called. In contrast, an instance of a generic subprogram is a (nongeneric) subprogram; hence, this instance can be called but it cannot be used to produce further instances. 3

12.1 Generic Declarations

A generic_declaration declares a generic unit, which is either a generic subprogram or a generic package. A generic_declaration includes a generic_formal_part declaring any generic formal parameters. A generic formal parameter can be an object; alternatively (unlike a parameter of a subprogram), it can be a type, a subprogram, or a package. 1

Syntax

generic_declaration ::= 2
  generic_subprogram_declaration | generic_package_declaration

generic_subprogram_declaration ::= 3/3
  generic_formal_part subprogram_specification
    [aspect_specification];

generic_package_declaration ::= 4
  generic_formal_part package_specification;

generic_formal_part ::= 5
  generic {generic_formal_parameter_declaration | use_clause}

generic_formal_parameter_declaration ::= 6
  formal_object_declaration
 | formal_type_declaration
 | formal_subprogram_declaration
 | formal_package_declaration

The only form of subtype_indication allowed within a generic_formal_part is a subtype_mark (that is, the subtype_indication shall not include an explicit constraint). The defining name of a generic subprogram shall be an identifier (not an operator_symbol). 7

Static Semantics

A generic_declaration declares a generic unit — a generic package, generic procedure, or generic function, as appropriate. 8/2

9 An entity is a *generic formal* entity if it is declared by a generic_formal_parameter_declaration. "Generic formal", or simply "formal", is used as a prefix in referring to objects, subtypes (and types), functions, procedures and packages, that are generic formal entities, as well as to their respective declarations. Examples: "generic formal procedure" or a "formal integer type declaration".

9.1/5 The list of generic_formal_parameter_declarations of a generic_formal_part form a *declaration list* of the generic unit.

Dynamic Semantics

10 The elaboration of a generic_declaration has no effect.

11 NOTE 1 Outside a generic unit a name that denotes the generic_declaration denotes the generic unit. In contrast, within the declarative region of the generic unit, a name that denotes the generic_declaration denotes the current instance.

12 NOTE 2 Within a generic subprogram_body, the name of this program unit acts as the name of a subprogram. Hence this name can be overloaded, and it can appear in a recursive call of the current instance. For the same reason, this name cannot appear after the reserved word **new** in a (recursive) generic_instantiation.

13/5 NOTE 3 A default_expression or default_name appearing in a generic_formal_part is not evaluated during elaboration of the generic_formal_part; instead, it is evaluated when used. However, the usual visibility rules apply to any name used in a default, with name resolution performed based on the location of the name within the generic_formal_part.

Examples

14 *Examples of generic formal parts:*

15
```
generic      -- parameterless
```

16
```
generic
   Size : Natural;  -- formal object
```

17
```
generic
   Length : Integer := 200;          -- formal object with a default expression
```

18
```
   Area   : Integer := Length*Length; -- formal object with a default expression
```

19
```
generic
   type Item  is private;                          -- formal type
   type Index is (<>);                             -- formal type
   type Row   is array(Index range <>) of Item;    -- formal type
   with function "<"(X, Y : Item) return Boolean;     -- formal subprogram
```

20 *Examples of generic declarations declaring generic subprograms Exchange and Squaring:*

21
```
generic
   type Elem is private;
procedure Exchange(U, V : in out Elem);
```

22/5
```
generic
   type Item (<>) is private;
   with function "*"(U, V : Item) return Item is <>;
function Squaring(X : Item) return Item;
```

23 *Example of a generic declaration declaring a generic package:*

24
```
generic
   type Item   is private;
   type Vector is array (Positive range <>) of Item;
   with function Sum(X, Y : Item) return Item;
package On_Vectors is
   function Sum  (A, B : Vector) return Vector;
   function Sigma(A    : Vector) return Item;
   Length_Error : exception;
end On_Vectors;
```

12.2 Generic Bodies

The body of a generic unit (a *generic body*) is a template for the instance bodies. The syntax of a generic body is identical to that of a nongeneric body. 1

Dynamic Semantics

The elaboration of a generic body has no other effect than to establish that the generic unit can from then on be instantiated without failing the Elaboration_Check. If the generic body is a child of a generic package, then its elaboration establishes that each corresponding declaration nested in an instance of the parent (see 10.1.1) can from then on be instantiated without failing the Elaboration_Check. 2

NOTE The syntax of generic subprograms implies that a generic subprogram body is always the completion of a declaration. 3

Examples

Example of a generic procedure body: 4

```
procedure Exchange(U, V : in out Elem) is  -- see 12.1
   T : Elem;  -- the generic formal type
begin
   T := U;
   U := V;
   V := T;
end Exchange;
```

5

Example of a generic function body: 6

```
function Squaring(X : Item) return Item is  -- see 12.1
begin
   return X*X;  -- the formal operator "*"
end Squaring;
```

7

Example of a generic package body: 8

```
package body On_Vectors is  -- see 12.1
```

9

```
   function Sum(A, B : Vector) return Vector is
      Result : Vector(A'Range); -- the formal type Vector
      Bias   : constant Integer := B'First - A'First;
   begin
      if A'Length /= B'Length then
         raise Length_Error;
      end if;

      for N in A'Range loop
         Result(N) := Sum(A(N), B(N + Bias)); -- the formal function Sum
      end loop;
      return Result;
   end Sum;

   function Sigma(A : Vector) return Item is
      Total : Item := A(A'First); -- the formal type Item
   begin
      for N in A'First + 1 .. A'Last loop
         Total := Sum(Total, A(N)); -- the formal function Sum
      end loop;
      return Total;
   end Sigma;
end On_Vectors;
```

10

11

12

12.3 Generic Instantiation

1 An instance of a generic unit is declared by a generic_instantiation.

Syntax

2/3
```
generic_instantiation ::=
    package defining_program_unit_name is
      new generic_package_name [generic_actual_part]
        [aspect_specification];
  | [overriding_indicator]
    procedure defining_program_unit_name is
      new generic_procedure_name [generic_actual_part]
        [aspect_specification];
  | [overriding_indicator]
    function defining_designator is
      new generic_function_name [generic_actual_part]
        [aspect_specification];
```

3
```
generic_actual_part ::=
  (generic_association {, generic_association})
```

4
```
generic_association ::=
  [generic_formal_parameter_selector_name =>] explicit_generic_actual_parameter
```

5
```
explicit_generic_actual_parameter ::= expression | variable_name
  | subprogram_name | entry_name | subtype_mark
  | package_instance_name
```

6 A generic_association is *named* or *positional* according to whether or not the *generic_formal_parameter_*selector_name is specified. Any positional associations shall precede any named associations.

7/5 The *generic actual parameter* is either the explicit_generic_actual_parameter given in a generic_association for each formal, or the corresponding default_expression, *default_*subtype_mark, or default_name if no generic_association is given for the formal. When the meaning is clear from context, the term "generic actual", or simply "actual", is used as a synonym for "generic actual parameter" and also for the view denoted by one, or the value of one.

Legality Rules

8 In a generic_instantiation for a particular kind of program unit (package, procedure, or function), the name shall denote a generic unit of the corresponding kind (generic package, generic procedure, or generic function, respectively).

9/3 The *generic_formal_parameter_*selector_name of a named generic_association shall denote a generic_formal_parameter_declaration of the generic unit being instantiated. If two or more formal subprograms have the same defining name, then named associations are not allowed for the corresponding actuals.

9.1/3 The generic_formal_parameter_declaration for a positional generic_association is the parameter with the corresponding position in the generic_formal_part of the generic unit being instantiated.

A generic_instantiation shall contain at most one generic_association for each formal. Each formal without an association shall have a default_expression, *default*_subtype_mark, or subprogram_default. 10/5

In a generic unit, Legality Rules are enforced at compile time of the generic_declaration and generic body, given the properties of the formals. In the visible part and formal part of an instance, Legality Rules are enforced at compile time of the generic_instantiation, given the properties of the actuals. In other parts of an instance, Legality Rules are not enforced; this rule does not apply when a given rule explicitly specifies otherwise. 11/5

Static Semantics

A generic_instantiation declares an instance; it is equivalent to the instance declaration (a package_-declaration or subprogram_declaration) immediately followed by the instance body, both at the place of the instantiation. 12

The instance is a copy of the text of the template. Each use of a formal parameter becomes (in the copy) a use of the actual, as explained below. An instance of a generic package is a package, that of a generic procedure is a procedure, and that of a generic function is a function. 13

The interpretation of each construct within a generic declaration or body is determined using the overloading rules when that generic declaration or body is compiled. In an instance, the interpretation of each (copied) construct is the same, except in the case of a name that denotes the generic_declaration or some declaration within the generic unit; the corresponding name in the instance then denotes the corresponding copy of the denoted declaration. The overloading rules do not apply in the instance. 14

In an instance, a generic_formal_parameter_declaration declares a view whose properties are identical to those of the actual, except when specified otherwise (in particular, see 6.1.1, “Preconditions and Postconditions”, 12.4, “Formal Objects”, and 12.6, “Formal Subprograms”). Similarly, for a declaration within a generic_formal_parameter_declaration, the corresponding declaration in an instance declares a view whose properties are identical to the corresponding declaration within the declaration of the actual. 15/5

Implicit declarations are also copied, and a name that denotes an implicit declaration in the generic denotes the corresponding copy in the instance. However, for a type declared within the visible part of the generic, a whole new set of primitive subprograms is implicitly declared for use outside the instance, and may differ from the copied set if the properties of the type in some way depend on the properties of some actual type specified in the instantiation. For example, if the type in the generic is derived from a formal private type, then in the instance the type will inherit subprograms from the corresponding actual type. 16

These new implicit declarations occur immediately after the type declaration in the instance, and override the copied ones. The copied ones can be called only from within the instance; the new ones can be called only from outside the instance, although for tagged types, the body of a new one can be executed by a call to an old one. 17

In the visible part of an instance, an explicit declaration overrides an implicit declaration if they are homographs, as described in 8.3. On the other hand, an explicit declaration in the private part of an instance overrides an implicit declaration in the instance, only if the corresponding explicit declaration in the generic overrides a corresponding implicit declaration in the generic. Corresponding rules apply to the other kinds of overriding described in 8.3. 18

Post-Compilation Rules

Recursive generic instantiation is not allowed in the following sense: if a given generic unit includes an instantiation of a second generic unit, then the instance generated by this instantiation shall not include an 19

instance of the first generic unit (whether this instance is generated directly, or indirectly by intermediate instantiations).

Dynamic Semantics

20 For the elaboration of a generic_instantiation, each generic_association is first evaluated. If a default is used, an implicit generic_association is assumed for this rule. These evaluations are done in an arbitrary order, except that the evaluation for a default actual takes place after the evaluation for another actual if the default includes a name that denotes the other one. Finally, the instance declaration and body are elaborated.

21 For the evaluation of a generic_association the generic actual parameter is evaluated. Additional actions are performed in the case of a formal object of mode **in** (see 12.4).

22 NOTE If a formal type is not tagged, then the type is treated as an untagged type within the generic body. Deriving from such a type in a generic body is permitted; the new type does not get a new tag value, even if the actual is tagged. Overriding operations for such a derived type cannot be dispatched to from outside the instance.

Examples

23 *Examples of generic instantiations (see 12.1):*

24/5
```
procedure Swap is new Exchange(Elem => Integer);
procedure Swap is new Exchange(Character);        -- Swap is overloaded
function Square is new Squaring(Integer);         -- "*" of Integer used by default
function Square1 is new Squaring(Item => Matrix, "*" => Matrix_Product);
function Square2 is new Squaring(Matrix, Matrix_Product); -- same as previous
```

25
```
package Int_Vectors is new On_Vectors(Integer, Table, "+");
```

26 *Examples of uses of instantiated units:*

27
```
Swap(A, B);
A := Square(A);
```

28
```
T : Table(1 .. 5) := (10, 20, 30, 40, 50);
N : Integer := Int_Vectors.Sigma(T);  -- 150
                                      -- (see 12.2 for the body of Sigma)
```

29
```
use Int_Vectors;
M : Integer := Sigma(T);  -- 150
```

12.4 Formal Objects

1 A generic formal object can be used to pass a value or variable to a generic unit.

Syntax

2/3
```
formal_object_declaration ::=
   defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
      [aspect_specification];
 | defining_identifier_list : mode access_definition [:= default_expression]
      [aspect_specification];
```

Name Resolution Rules

3 The expected type for the default_expression, if any, of a formal object is the type of the formal object.

4 For a generic formal object of mode **in**, the expected type for the actual is the type of the formal.

5/2 For a generic formal object of mode **in out**, the type of the actual shall resolve to the type determined by the subtype_mark, or for a formal_object_declaration with an access_definition, to a specific anonymous

access type. If the anonymous access type is an access-to-object type, the type of the actual shall have the same designated type as that of the access_definition. If the anonymous access type is an access-to-subprogram type, the type of the actual shall have a designated profile which is type conformant with that of the access_definition.

Legality Rules

If a generic formal object has a default_expression, then the mode shall be **in** (either explicitly or by default); otherwise, its mode shall be either **in** or **in out**. 6

For a generic formal object of mode **in**, the actual shall be an expression. For a generic formal object of mode **in out**, the actual shall be a name that denotes a variable for which renaming is allowed (see 8.5.1). 7

In the case where the type of the formal is defined by an access_definition, the type of the actual and the type of the formal: 8/2

- shall both be access-to-object types with statically matching designated subtypes and with both or neither being access-to-constant types; or 8.1/2
- shall both be access-to-subprogram types with subtype conformant designated profiles. 8.2/2

For a formal_object_declaration of mode **in out** with a null_exclusion or an access_definition that has a null_exclusion, the subtype of the actual matching the formal_object_declaration shall exclude null. In addition, if the actual matching the formal_object_declaration statically denotes the generic formal object of mode **in out** of another generic unit *G*, and the instantiation containing the actual occurs within the body of *G* or within the body of a generic unit declared within the declarative region of *G*, then the declaration of the formal object of *G* shall have a null_exclusion. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 8.3/5

Static Semantics

A formal_object_declaration declares a generic formal object. The default mode is **in**. For a formal object of mode **in**, the nominal subtype is the one denoted by the subtype_mark or access_definition in the declaration of the formal. For a formal object of mode **in out**, its type is determined by the subtype_mark or access_definition in the declaration; its nominal subtype is nonstatic, even if the subtype_mark denotes a static subtype; for a composite type, its nominal subtype is unconstrained if the first subtype of the type is unconstrained, even if the subtype_mark denotes a constrained subtype. 9/2

In an instance, a formal_object_declaration of mode **in** is a *full constant declaration* and declares a new stand-alone constant object whose initialization expression is the actual, whereas a formal_object_declaration of mode **in out** declares a view whose properties are identical to those of the actual. 10/2

Dynamic Semantics

For the evaluation of a generic_association for a formal object of mode **in**, a constant object is created, the value of the actual parameter is converted to the nominal subtype of the formal object, and assigned to the object, including any value adjustment — see 7.6. 11

NOTE The constraints that apply to a generic formal object of mode **in out** are those of the corresponding generic actual parameter (not those implied by the subtype_mark that appears in the formal_object_declaration). Therefore, to avoid confusion, it is recommended that the name of a first subtype be used for the declaration of such a formal object. 12

12.5 Formal Types

1/2 A generic formal subtype can be used to pass to a generic unit a subtype whose type is in a certain category of types.

Syntax

2/3
```
formal_type_declaration ::=
     formal_complete_type_declaration
   | formal_incomplete_type_declaration
```

2.1/5
```
formal_complete_type_declaration ::=
   type defining_identifier[discriminant_part] is formal_type_definition
     [or use default_subtype_mark] [aspect_specification];
```

2.2/5
```
formal_incomplete_type_declaration ::=
   type defining_identifier[discriminant_part] [is tagged]
     [or use default_subtype_mark];
```

3/2
```
formal_type_definition ::=
     formal_private_type_definition
   | formal_derived_type_definition
   | formal_discrete_type_definition
   | formal_signed_integer_type_definition
   | formal_modular_type_definition
   | formal_floating_point_definition
   | formal_ordinary_fixed_point_definition
   | formal_decimal_fixed_point_definition
   | formal_array_type_definition
   | formal_access_type_definition
   | formal_interface_type_definition
```

Legality Rules

4 For a generic formal subtype, the actual shall be a subtype_mark; it denotes the *(generic) actual subtype*.

Static Semantics

5 A formal_type_declaration declares a *(generic) formal type*, and its first subtype, the *(generic) formal subtype*.

6/3 The form of a formal_type_definition *determines a category (of types)* to which the formal type belongs. For a formal_private_type_definition the reserved words **tagged** and **limited** indicate the category of types (see 12.5.1). The reserved word **tagged** also plays this role in the case of a formal_incomplete_type_declaration. For a formal_derived_type_definition the category of types is the derivation class rooted at the ancestor type. For other formal types, the name of the syntactic category indicates the category of types; a formal_discrete_type_definition defines a discrete type, and so on.

Legality Rules

7/2 The actual type shall be in the category determined for the formal.

7.1/5 The *default*_subtype_mark, if any, shall denote a subtype which is allowed as an actual subtype for the formal type.

Static Semantics

The formal type also belongs to each category that contains the determined category. The primitive subprograms of the type are as for any type in the determined category. For a formal type other than a formal derived type, these are the predefined operators of the type. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later immediately within the declarative region in which the type is declared according to the rules of 7.3.1. In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type and even if it is never declared for the actual type, unless the actual type is an untagged record type, in which case it declares a view of the primitive (equality) operator. The rules specific to formal derived types are given in 12.5.1. 8/5

NOTE 1 Generic formal types, like all types, are not named. Instead, a name can denote a generic formal subtype. Within a generic unit, a generic formal type is considered as being distinct from all other (formal or nonformal) types. 9

NOTE 2 A discriminant_part is allowed only for certain kinds of types, and therefore only for certain kinds of generic formal types. See 3.7. 10

Examples

Examples of generic formal types: 11

```
type Item is private;                                                    12
type Buffer(Length : Natural) is limited private;

type Enum  is (<>);                                                      13
type Int   is range <>;
type Angle is delta <>;
type Mass  is digits <>;

type Table is array (Enum) of Item;                                      14
```

Example of a generic formal part declaring a formal integer type: 15

```
generic                                                                  16
   type Rank is range <>;
   First  : Rank := Rank'First;
   Second : Rank := First + 1;  -- the operator "+" of the type Rank
```

12.5.1 Formal Private and Derived Types

In its most general form, the category determined for a formal private type is all types, but the category can be restricted to only nonlimited types or to only tagged types. Similarly, the category for a formal incomplete type is all types but the category can be restricted to only tagged types; unlike other formal types, the actual type can be incompletely defined, and not ready to be frozen (see 13.14). The category determined for a formal derived type is the derivation class rooted at the ancestor type. 1/5

Syntax

formal_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] **private** 2

formal_derived_type_definition ::= 3/5
   [**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list] **with private**]

Legality Rules

If a generic formal type declaration has a known_discriminant_part, then it shall not include a default_expression for a discriminant. 4

5/3 The *ancestor subtype* of a formal derived type is the subtype denoted by the subtype_mark of the formal_derived_type_definition. For a formal derived type declaration, the reserved words **with private** shall appear if and only if the ancestor type is a tagged type; in this case the formal derived type is a private extension of the ancestor type and the ancestor shall not be a class-wide type. Similarly, an interface_list or the optional reserved words **abstract** or **synchronized** shall appear only if the ancestor type is a tagged type. The reserved word **limited** or **synchronized** shall appear only if the ancestor type and any progenitor types are limited types. The reserved word **synchronized** shall appear (rather than **limited**) if the ancestor type or any of the progenitor types are synchronized interfaces. The ancestor type shall be a limited interface if the reserved word **synchronized** appears.

5.1/5 The actual type for a formal derived type shall be a descendant of the ancestor type and every progenitor of the formal type. The actual type for a formal derived type shall be tagged if and only if the formal derived type is a private extension. If the reserved word **synchronized** appears in the declaration of the formal derived type, the actual type shall be a synchronized tagged type.

6/5 If a formal private or derived subtype is definite, then the actual subtype shall also be definite. If the formal type is nonlimited, the actual type shall be nonlimited.

6.1/3 A formal_incomplete_type_declaration declares a formal incomplete type. The only view of a formal incomplete type is an incomplete view. Thus, a formal incomplete type is subject to the same usage restrictions as any other incomplete type — see 3.10.1.

7/5 For a generic formal derived type with no discriminant_part, the actual subtype shall be statically compatible with the ancestor subtype. Furthermore:

8/5 • If the ancestor subtype is constrained, the actual subtype shall be constrained;

9 • If the ancestor subtype is an unconstrained access or composite subtype, the actual subtype shall be unconstrained.

10 • If the ancestor subtype is an unconstrained discriminated subtype, then the actual shall have the same number of discriminants, and each discriminant of the actual shall correspond to a discriminant of the ancestor, in the sense of 3.7.

10.1/2 • If the ancestor subtype is an access subtype, the actual subtype shall exclude null if and only if the ancestor subtype excludes null.

11/3 The declaration of a formal derived type shall not have a known_discriminant_part. For a generic formal private or incomplete type with a known_discriminant_part:

12 • The actual type shall be a type with the same number of discriminants.

13 • The actual subtype shall be unconstrained.

14 • The subtype of each discriminant of the actual type shall statically match the subtype of the corresponding discriminant of the formal type.

15/5 For a generic formal type with an unknown_discriminant_part, the actual may have discriminants, though that is not required, and may be definite or indefinite.

15.1/4 When enforcing Legality Rules, for the purposes of determining within a generic body whether a type is unconstrained in any partial view, a discriminated subtype is considered to have a constrained partial view if it is a descendant of an untagged generic formal private or derived type.

Static Semantics

16/2 The category determined for a formal private type is as follows:

17/2 *Type Definition* *Determined Category*

limited private	the category of all types
private	the category of all nonlimited types
tagged limited private	the category of all tagged types
tagged private	the category of all nonlimited tagged types

The presence of the reserved word **abstract** determines whether the actual type may be abstract. 18

The category determined for a formal incomplete type is the category of all types, unless the formal_type_declaration includes the reserved word **tagged**; in this case, it is the category of all tagged types. 18.1/3

A formal private or derived type is a private or derived type, respectively. A formal derived tagged type is a private extension. A formal private or derived type is abstract if the reserved word **abstract** appears in its declaration. 19

For a formal derived type, the characteristics (including components, but excluding discriminants if there is a new discriminant_part), predefined operators, and inherited user-defined primitive subprograms are determined by its ancestor type and its progenitor types (if any), in the same way that those of a derived type are determined by those of its parent type and its progenitor types (see 3.4 and 7.3.1). 20/3

In an instance, the copy of an implicit declaration of a primitive subprogram of a formal derived type declares a view of the corresponding primitive subprogram of the ancestor or progenitor of the formal derived type, even if this primitive has been overridden for the actual type and even if it is never declared for the actual type. When the ancestor or progenitor of the formal derived type is itself a formal type, the copy of the implicit declaration declares a view of the corresponding copied operation of the ancestor or progenitor. In the case of a formal private extension, however, the tag of the formal type is that of the actual type, so if the tag in a call is statically determined to be that of the formal type, the body executed will be that corresponding to the actual type. 21/3

In an instance, the implicitly composed and additive aspects (see 13.1.1) of a formal type are those of the actual; for a nonoverridable aspect, a formal derived type inherits the aspect if the ancestor or any progenitor has the aspect, according to the rules given in 13.1. 21.1/5

For a prefix S that denotes a formal indefinite subtype, the following attribute is defined: 22/1

S'Definite S'Definite yields True if the actual subtype corresponding to S is definite; otherwise, it yields False. The value of this attribute is of the predefined type Boolean. 23/3

Dynamic Semantics

In the case where a formal type has unknown discriminants, and the actual type is a class-wide type *T*'Class: 23.1/3

- For the purposes of defining the primitive operations of the formal type, each of the primitive operations of the actual type is considered to be a subprogram as described in 3.4.1 for primitive operations of a class-wide type. 23.2/6
- If the corresponding operation of *T* has no controlling formal parameters, then the controlling tag value is determined by the context of the call, according to the rules for tag-indeterminate calls (see 3.9.2 and 5.2). In the case where the tag would be statically determined to be that of the formal type, the call raises Program_Error. If such a function is renamed, any call on the renaming raises Program_Error. 23.3/2

NOTE 1 The actual type can be abstract only if the formal type is abstract (see 3.9.3). 27

28/5 NOTE 2 If the formal has a discriminant_part, the actual can be either definite or indefinite. Otherwise, the actual can only be definite.

12.5.2 Formal Scalar Types

1/2 A *formal scalar type* is one defined by any of the formal_type_definitions in this subclause. The category determined for a formal scalar type is the category of all discrete, signed integer, modular, floating point, ordinary fixed point, or decimal types.

Syntax

2 formal_discrete_type_definition ::= (<>)

3 formal_signed_integer_type_definition ::= **range** <>

4 formal_modular_type_definition ::= **mod** <>

5 formal_floating_point_definition ::= **digits** <>

6 formal_ordinary_fixed_point_definition ::= **delta** <>

7 formal_decimal_fixed_point_definition ::= **delta** <> **digits** <>

Legality Rules

8 The actual type for a formal scalar type shall not be a nonstandard numeric type.

12.5.3 Formal Array Types

1/2 The category determined for a formal array type is the category of all array types.

Syntax

2 formal_array_type_definition ::= array_type_definition

Legality Rules

3 The only form of discrete_subtype_definition that is allowed within the declaration of a generic formal (constrained) array subtype is a subtype_mark.

4 For a formal array subtype, the actual subtype shall satisfy the following conditions:

5 • The formal array type and the actual array type shall have the same dimensionality; the formal subtype and the actual subtype shall be either both constrained or both unconstrained.

6 • For each index position, the index types shall be the same, and the index subtypes (if unconstrained), or the index ranges (if constrained), shall statically match (see 4.9.1).

7 • The component subtypes of the formal and actual array types shall statically match.

8 • If the formal type has aliased components, then so shall the actual.

Examples

9 *Example of formal array types:*

10 -- *given the generic package*

```
generic                                                                    11
   type Item   is private;
   type Index  is (<>);
   type Vector is array (Index range <>) of Item;
   type Table  is array (Index) of Item;
package P is
   ...
end P;
```

```
-- and the types                                                           12
```

```
type Mix    is array (Color range <>) of Boolean;                          13
type Option is array (Color) of Boolean;
```

```
-- then Mix can match Vector and Option can match Table                    14
```

```
package R is new P(Item   => Boolean, Index => Color,                      15
                   Vector => Mix,     Table => Option);
```

```
-- Note that Mix cannot match Table and Option cannot match Vector         16
```

12.5.4 Formal Access Types

The category determined for a formal access type is the category of all access types. 1/2

Syntax

formal_access_type_definition ::= access_type_definition 2

Legality Rules

For a formal access-to-object type, the designated subtypes of the formal and actual types shall statically match. 3

If and only if the general_access_modifier **constant** applies to the formal, the actual shall be an access-to-constant type. If the general_access_modifier **all** applies to the formal, then the actual shall be a general access-to-variable type (see 3.10). If and only if the formal subtype excludes null, the actual subtype shall exclude null. 4/2

For a formal access-to-subprogram subtype, the designated profiles of the formal and the actual shall be subtype conformant. 5/3

Examples

Example of formal access types: 6

```
-- the formal types of the generic package                                 7
```

```
generic                                                                    8
   type Node is private;
   type Link is access Node;
package P is
   ...
end P;
```

```
-- can be matched by the actual types                                      9
```

```
type Car;                                                                  10
type Car_Name is access Car;
```

```
type Car is                                                                11
   record
      Pred, Succ : Car_Name;
      Number     : License_Number;
      Owner      : Person;
   end record;
```

```
-- in the following generic instantiation                                  12
```

13
```
package R is new P(Node => Car, Link => Car_Name);
```

12.5.5 Formal Interface Types

1/2 The category determined for a formal interface type is the category of all interface types.

Syntax

2/2 formal_interface_type_definition ::= interface_type_definition

Legality Rules

3/2 The actual type shall be a descendant of every progenitor of the formal type.

4/2 The actual type shall be a limited, task, protected, or synchronized interface if and only if the formal type is also, respectively, a limited, task, protected, or synchronized interface.

Examples

5/5 *Example of the use of a generic with a formal interface type, to establish a standard interface that all tasks will implement so they can be managed appropriately by an application-specific scheduler:*

6/2
```
type Root_Work_Item is tagged private;
```

7/2
```
generic
   type Managed_Task is task interface;
   type Work_Item(<>) is new Root_Work_Item with private;
package Server_Manager is
   task type Server is new Managed_Task with
      entry Start(Data : in out Work_Item);
   end Server;
end Server_Manager;
```

12.6 Formal Subprograms

1 Formal subprograms can be used to pass callable entities to a generic unit.

Syntax

2/2 formal_subprogram_declaration ::= formal_concrete_subprogram_declaration
| formal_abstract_subprogram_declaration

2.1/3 formal_concrete_subprogram_declaration ::=
with subprogram_specification [**is** subprogram_default]
[aspect_specification];

2.2/3 formal_abstract_subprogram_declaration ::=
with subprogram_specification **is abstract** [subprogram_default]
[aspect_specification];

3/2 subprogram_default ::= default_name | <> | **null**

4 default_name ::= name

4.1/2 A subprogram_default of **null** shall not be specified for a formal function or for a formal_abstract_subprogram_declaration.

Name Resolution Rules

5 The expected profile for the default_name, if any, is that of the formal subprogram.

For a generic formal subprogram, the expected profile for the actual is that of the formal subprogram. 6

Legality Rules

The profiles of the formal and any named default shall be mode conformant. 7/3

The profiles of the formal and actual shall be mode conformant. 8/3

For a parameter or result subtype of a formal_subprogram_declaration that has an explicit null_exclusion: 8.1/2

- if the actual matching the formal_subprogram_declaration statically denotes a generic formal subprogram of another generic unit *G*, and the instantiation containing the actual occurs within the body of a generic unit *G* or within the body of a generic unit declared within the declarative region of the generic unit *G*, then the corresponding parameter or result type of the formal subprogram of *G* shall have a null_exclusion; 8.2/5
- otherwise, the subtype of the corresponding parameter or result type of the actual matching the formal_subprogram_declaration shall exclude null. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit. 8.3/2

If the named default, if any, is a prefixed view, the prefix of that view shall denote an object for which renaming is allowed (see 8.5.1). Similarly, if the actual subprogram in an instantiation is a prefixed view, the prefix of that view shall denote an object for which renaming is allowed. 8.4/5

If a formal parameter of a formal_abstract_subprogram_declaration is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a *controlling type* of the formal_abstract_subprogram_declaration. Similarly, if the result of a formal_abstract_subprogram_declaration for a function is of a specific tagged type *T* or of an anonymous access type designating a specific tagged type *T*, *T* is called a controlling type of the formal_abstract_subprogram_declaration. A formal_abstract_subprogram_declaration shall have exactly one controlling type, and that type shall not be incomplete. 8.5/5

The actual subprogram for a formal_abstract_subprogram_declaration shall be: 8.6/5

- a dispatching operation of the controlling type; or 8.7/5
- if the controlling type is a formal type, and the actual type corresponding to that formal type is a specific type *T*, a dispatching operation of type *T*; or 8.8/5
- if the controlling type is a formal type, and the actual type is a class-wide type *T*'Class, an implicitly declared subprogram corresponding to a primitive operation of type *T* (see Static Semantics below). 8.9/5

Static Semantics

A formal_subprogram_declaration declares a generic formal subprogram. The types of the formal parameters and result, if any, of the formal subprogram are those determined by the subtype_marks given in the formal_subprogram_declaration; however, independent of the particular subtypes that are denoted by the subtype_marks, the nominal subtypes of the formal parameters and result, if any, are defined to be nonstatic, and unconstrained if of an array type (no applicable index constraint is provided in a call on a formal subprogram). In an instance, a formal_subprogram_declaration declares a view of the actual. The profile of this view takes its subtypes and calling convention from the original profile of the actual entity, while taking the formal parameter names and default_expressions from the profile given in the formal_subprogram_declaration. The view is a function or procedure, never an entry. 9

If a subtype_mark in the profile of the formal_subprogram_declaration denotes a formal private or formal derived type and the actual type for this formal type is a class-wide type *T*'Class, then for the 9.1/3

purposes of resolving the corresponding actual subprogram at the point of the instantiation, certain implicit declarations may be available as possible resolutions as follows:

9.2/6 For each primitive subprogram of *T* that is directly visible at the point of the instantiation, and that has at least one controlling formal parameter, a corresponding implicitly declared subprogram with the same defining name, and having the same profile as the primitive subprogram except that *T* is systematically replaced by *T*'Class in the types of its profile, is potentially use-visible. The body of such a subprogram is as defined in 3.4.1 for primitive subprograms of a class-wide type.

10 If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal.

10.1/2 If a generic unit has a subprogram_default specified by the reserved word **null**, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a null procedure having the profile given in the formal_subprogram_declaration.

10.2/2 The subprogram declared by a formal_abstract_subprogram_declaration with a controlling type *T* is a dispatching operation of type *T*.

11/5 NOTE 1 The matching rules for formal subprograms state requirements that are similar to those applying to subprogram_renaming_declarations (see 8.5.4). In particular, the name of a parameter of the formal subprogram can be different from that of the corresponding parameter of the actual subprogram; similarly, for these parameters, default_expressions can be different.

12 NOTE 2 The constraints that apply to a parameter of a formal subprogram are those of the corresponding formal parameter of the matching actual subprogram (not those implied by the corresponding subtype_mark in the _specification of the formal subprogram). A similar remark applies to the result of a function. Therefore, to avoid confusion, it is recommended that the name of a first subtype be used in any declaration of a formal subprogram.

13 NOTE 3 The subtype specified for a formal parameter of a generic formal subprogram can be any visible subtype, including a generic formal subtype of the same generic_formal_part.

14 NOTE 4 A formal subprogram is matched by an attribute of a type if the attribute is a function with a matching specification. An enumeration literal of a given type matches a parameterless formal function whose result type is the given type.

15/5 NOTE 5 A default_name denotes an entity that is visible or directly visible at the place of the generic_declaration; a box used as a default is equivalent to a name that denotes an entity that is directly visible at the place of the generic_instantiation.

16/2 NOTE 6 The actual subprogram cannot be abstract unless the formal subprogram is a formal_abstract_subprogram_declaration (see 3.9.3).

16.1/5 NOTE 7 The subprogram declared by a formal_abstract_subprogram_declaration is an abstract subprogram. All calls on a subprogram declared by a formal_abstract_subprogram_declaration are limited to dispatching calls. See 3.9.3.

16.2/2 NOTE 8 A null procedure as a subprogram default has convention Intrinsic (see 6.3.1).

Examples

17 *Examples of generic formal subprograms:*

18/2
```
with function "+"(X, Y : Item) return Item is <>;
with function Image(X : Enum) return String is Enum'Image;
with procedure Update is Default_Update;
with procedure Pre_Action(X : in Item) is null;  -- defaults to no action
with procedure Write(S    : not null access Root_Stream_Type'Class;
                     Desc : Descriptor)
                     is abstract Descriptor'Write;  -- see 13.13.2
-- Dispatching operation on Descriptor with default
```

19
```
-- given the generic procedure declaration
```

```
generic                                                                    20
   with procedure Action (X : in Item);
procedure Iterate(Seq : in Item_Sequence);
-- and the procedure                                                       21
procedure Put_Item(X : in Item);                                           22
-- the following instantiation is possible                                 23
procedure Put_List is new Iterate(Action => Put_Item);                     24
```

12.7 Formal Packages

Formal packages can be used to pass packages to a generic unit. The formal_package_declaration declares that the formal package is an instance of a given generic package. Upon instantiation, the actual package has to be an instance of that generic package. 1

Syntax

formal_package_declaration ::= 2/3
   with package defining_identifier **is new** *generic_package_*name formal_package_actual_part
      [aspect_specification];

formal_package_actual_part ::= 3/2
   ([**others** =>] <>)
  | [generic_actual_part]
  | (formal_package_association {, formal_package_association} [, **others** => <>])

formal_package_association ::= 3.1/2
   generic_association
  | *generic_formal_parameter_*selector_name => <>

Any positional formal_package_associations shall precede any named formal_package_associations. 3.2/2

Legality Rules

The *generic_package_*name shall denote a generic package (the *template* for the formal package); the formal package is an instance of the template. 4

The *generic_formal_parameter_*selector_name of a formal_package_association shall denote a generic_formal_parameter_declaration of the template. If two or more formal subprograms of the template have the same defining name, then named associations are not allowed for the corresponding actuals. 4.1/3

A formal_package_actual_part shall contain at most one formal_package_association for each formal parameter. If the formal_package_actual_part does not include "**others** => <>", each formal parameter without an association shall have a default_expression or subprogram_default. 4.2/3

The rules for matching between formal_package_associations and the generic formals of the template are as follows: 4.3/3

- If all of the formal_package_associations are given by generic associations, the explicit_generic_actual_parameters of the formal_package_associations shall be legal for an instantiation of the template. 4.4/3

4.5/5 • If a formal_package_association for a formal type *T* of the template is given by <>, then the formal_package_association for any other generic_formal_parameter_declaration of the template that mentions *T* directly or indirectly shall also be given by <>.

5/2 The actual shall be an instance of the template. If the formal_package_actual_part is (<>) or (**others** => <>), then the actual may be any instance of the template; otherwise, certain of the actual parameters of the actual instance shall match the corresponding actual parameters of the formal package, determined as follows:

5.1/2 • If the formal_package_actual_part includes generic_associations as well as associations with <>, then only the actual parameters specified explicitly with generic_associations are required to match;

5.2/2 • Otherwise, all actual parameters shall match, whether any actual parameter is given explicitly or by default.

5.3/2 The rules for matching of actual parameters between the actual instance and the formal package are as follows:

6/2 • For a formal object of mode **in**, the actuals match if they are static expressions with the same value, or if they statically denote the same constant, or if they are both the literal **null**.

7 • For a formal subtype, the actuals match if they denote statically matching subtypes.

8 • For other kinds of formals, the actuals match if they statically denote the same entity.

8.1/1 For the purposes of matching, any actual parameter that is the name of a formal object of mode **in** is replaced by the formal object's actual expression (recursively).

Static Semantics

9 A formal_package_declaration declares a generic formal package.

10/2 The visible part of a formal package includes the first list of basic_declarative_items of the package_specification. In addition, for each actual parameter that is not required to match, a copy of the declaration of the corresponding formal parameter of the template is included in the visible part of the formal package. If the copied declaration is for a formal type, copies of the implicit declarations of the primitive subprograms of the formal type are also included in the visible part of the formal package.

11/2 For the purposes of matching, if the actual instance *A* is itself a formal package, then the actual parameters of *A* are those specified explicitly or implicitly in the formal_package_actual_part for *A*, plus, for those not specified, the copies of the formal parameters of the template included in the visible part of *A*.

Examples

12/2 *Example of a generic package with formal package parameters:*

13/2
```
with Ada.Containers.Ordered_Maps;  -- see A.18.6
generic
   with package Mapping_1 is new Ada.Containers.Ordered_Maps(<>);
   with package Mapping_2 is new Ada.Containers.Ordered_Maps
                                    (Key_Type => Mapping_1.Element_Type,
                                     others => <>);
package Ordered_Join is
   -- Provide a "join" between two mappings
14/2    subtype Key_Type is Mapping_1.Key_Type;
   subtype Element_Type is Mapping_2.Element_Type;
15/2    function Lookup(Key : Key_Type) return Element_Type;
16/2    ...
end Ordered_Join;
```

Example of an instantiation of a package with formal packages: 17/2

```
with Ada.Containers.Ordered_Maps;                                                    18/2
package Symbol_Package is

   subtype Key_String is String(1..5);                                               19/5
   type String_Id is ...

   type Symbol_Info is ...                                                           20/2

   package String_Table is new Ada.Containers.Ordered_Maps                           21/5
           (Key_Type => Key_String,
            Element_Type => String_Id);

   package Symbol_Table is new Ada.Containers.Ordered_Maps                           22/2
           (Key_Type => String_Id,
            Element_Type => Symbol_Info);

   package String_Info is new Ordered_Join(Mapping_1 => String_Table,                23/2
                                           Mapping_2 => Symbol_Table);

   Apple_Info : constant Symbol_Info := String_Info.Lookup("Apple");                 24/2

end Symbol_Package;                                                                  25/2
```

12.8 Example of a Generic Package

Examples

The following example provides a possible formulation of stacks by means of a generic package. The size of each stack and the type of the stack elements are provided as generic formal parameters. 1

This paragraph was deleted. 2/1

```
generic                                                                              3
   Size : Positive;
   type Item is private;
package Stack is
   procedure Push(E : in  Item);
   procedure Pop (E : out Item);
   Overflow, Underflow : exception;
end Stack;

package body Stack is                                                                4

   type Table is array (Positive range <>) of Item;                                  5
   Space : Table(1 .. Size);
   Index : Natural := 0;

   procedure Push(E : in Item) is                                                    6
   begin
      if Index >= Size then
         raise Overflow;
      end if;
      Index := Index + 1;
      Space(Index) := E;
   end Push;

   procedure Pop(E : out Item) is                                                    7
   begin
      if Index = 0 then
         raise Underflow;
      end if;
      E := Space(Index);
      Index := Index - 1;
   end Pop;

end Stack;                                                                           8
```

9 Instances of this generic package can be obtained as follows:

10
```
package Stack_Int  is new Stack(Size => 200, Item => Integer);
package Stack_Bool is new Stack(100, Boolean);
```

11 Thereafter, the procedures of the instantiated packages can be called as follows:

12
```
Stack_Int.Push(N);
Stack_Bool.Push(True);
```

13 Alternatively, a generic formulation of the type Stack can be given as follows (package body omitted):

14
```
generic
   type Item is private;
package On_Stacks is
   type Stack(Size : Positive) is limited private;
   procedure Push(S : in out Stack; E : in  Item);
   procedure Pop (S : in out Stack; E : out Item);
   Overflow, Underflow : exception;
private
   type Table is array (Positive range <>) of Item;
   type Stack(Size : Positive) is
      record
         Space : Table(1 .. Size);
         Index : Natural := 0;
      end record;
end On_Stacks;
```

15 In order to use such a package, an instance has to be created and thereafter stacks of the corresponding type can be declared:

16
```
declare
   package Stack_Real is new On_Stacks(Real); use Stack_Real;
   S : Stack(100);
begin
   ...
   Push(S, 2.54);
   ...
end;
```

13 Representation Issues

This clause describes features for querying and controlling certain aspects of entities and for interfacing to hardware. 1/3

13.1 Operational and Representation Aspects

Two kinds of aspects of entities can be specified: representation aspects and operational aspects. Representation aspects affect how the types and other entities of the language are to be mapped onto the underlying machine. Operational aspects determine other properties of entities. When not specified otherwise, an aspect is defined to be an operational aspect. 0.1/6

Either kind of aspect of an entity may be specified by means of an aspect_specification (see 13.1.1), which is an optional element of most kinds of declarations and applies to the entity or entities being declared. Aspects may also be specified by certain other constructs occurring subsequent to the declaration of the affected entity: a representation aspect value may be specified by means of a representation item and an operational aspect value may be specified by means of an operational item. 0.2/3

There are six kinds of *representation items*: attribute_definition_clauses for representation attributes, enumeration_representation_clauses, record_representation_clauses, at_clauses, component_clauses, and *representation pragmas*. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware). 1/1

An *operational item* is an attribute_definition_clause for an operational attribute. 1.1/1

An operational item or a representation item applies to an entity identified by a local_name, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation. 1.2/1

Syntax

aspect_clause ::= attribute_definition_clause 2/1
| enumeration_representation_clause
| record_representation_clause
| at_clause

local_name ::= direct_name 3
| direct_name'attribute_designator
| *library_unit*_name

A representation pragma is an aspect-related pragma (see 2.8), and is allowed only at places where an aspect_clause or compilation_unit is allowed. 4/6

Name Resolution Rules

In an operational item or representation item, if the local_name is a direct_name, then it shall resolve to denote a declaration (or, in the case of a pragma, one or more declarations) that occurs immediately within the same declarative region as the item. If the local_name has an attribute_designator, then it shall resolve to denote an implementation-defined component (see 13.5.1) or a class-wide type implicitly declared immediately within the same declarative region as the item. A local_name that is a *library_unit*_name (only permitted in a representation pragma) shall resolve to denote the library_item that immediately precedes (except for other pragmas) the representation pragma. 5/1

Legality Rules

6/1 The local_name of an aspect_clause or representation pragma shall statically denote an entity (or, in the case of a pragma, one or more entities) declared immediately preceding it in a compilation, or within the same declarative_part, package_specification, task_definition, protected_definition, or record_definition as the representation or operational item. If a local_name denotes a local callable entity, it may do so through a local subprogram_renaming_declaration (as a way to resolve ambiguity in the presence of overloading); otherwise, the local_name shall not denote a renaming_declaration.

7/5 The *representation* of an object consists of a certain number of bits (the *size* of the object). For an object of an elementary type, these are the bits that are normally read or updated by the machine code when loading, storing, or operating-on the value of the object. For an object of a composite type, these are the bits reserved for this object, and include bits occupied by subcomponents of the object. If the size of an object is greater than that of its subtype, the additional bits are padding bits. For an elementary object, these padding bits are normally read and updated along with the others. For a composite object, it is unspecified whether padding bits are read or updated in any given composite operation.

8/5 A representation item *directly specifies* a *representation aspect* of the entity denoted by the local_name, except in the case of a type-related representation item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the subtype's type. A representation item that names a subtype is either *subtype-specific* (Size, Object_Size, and Alignment clauses) or *type-related* (all others).

8.1/3 An operational item *directly specifies* an *operational aspect* of the entity denoted by the local_name, except in the case of a type-related operational item, whose local_name shall denote a first subtype, and which directly specifies an aspect of the type of the subtype.

8.2/5 Aspects that can be specified for types and subtypes are also classified into type-related or subtype-specific aspects. Representation aspects that can be specified for types and subtypes are considered type-related unless specified otherwise. In contrast, the classification of operational aspects are given with the definition of the aspect. Type-related aspects have the same value for all subtypes of (a view of) a type, while subtype-specific aspects may differ for different subtypes of the same type.

9/5 A representation item or operational item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.14).

9.1/5 A representation aspect of a subtype or type shall not be specified (whether by a representation item or an aspect_specification) before the type is completely defined (see 3.11.1).

9.2/5 If a representation item, operational item, library unit pragma (see J.15), or aspect_specification is given that directly specifies an aspect of an entity, then it is illegal to give another representation item, operational item, library unit pragma, or aspect_specification that directly specifies the same aspect of the entity.

9.3/5 Unless otherwise specified, it is illegal to specify an operational or representation aspect of a generic formal parameter.

10/5 A *by-reference primitive* is a user-defined primitive subprogram for a type *T* that has an access result designating type *T*, or that has a formal parameter that is an access parameter designating type *T* or is aliased and of type *T*. It is illegal to specify a nonconfirming type-related representation aspect for an untagged type *T* if it is derived from a by-reference type or inherits one or more by-reference primitives, or if one or more types have been derived from *T* prior to the specification of the aspect and type *T* is a by-reference type or defines one or more by-reference primitives that are inherited by these descendants.

If a type-related aspect is defined for the partial view of a type, then it has the same definition for the full view of the type, except for certain Boolean-valued operational aspects where the language specifies that the partial view can have the value False even when the full view has the value True. Type-related aspects cannot be specified, and are not defined for an incomplete view of a type. Representation aspects of a generic formal parameter are the same as those of the actual. Specification of a type-related representation aspect is not allowed for a descendant of a generic formal untagged type. 11/5

The specification of the Size aspect for a given subtype, or the size or storage place for an object (including a component) of a given subtype, shall allow for enough storage space to accommodate any value of the subtype. 12/3

The specification of certain language-defined aspects is not required to be supported by all implementations; in such an implementation, the specification for such an aspect is illegal or raises an exception at run time. 13/5

A type_declaration is illegal if it has one or more progenitors, and a nonconfirming value was specified for a representation aspect of an ancestor, and this conflicts with the representation of some other ancestor. The cases that cause conflicts are implementation defined. 13.1/3

When specifying an aspect that denotes a subprogram, the profile of the subprogram shall be mode conformant with the one required for the aspect, and the convention shall be Ada. Additional requirements are defined for particular aspects. 13.2/5

Static Semantics

If two subtypes statically match, then their subtype-specific aspects (for example, Size and Alignment) are the same. 14/5

A derived type inherits each type-related representation aspect of its parent type that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent type from the grandparent type. A derived subtype inherits each subtype-specific representation aspect of its parent subtype that was directly specified before the declaration of the derived type, or (in the case where the parent is derived) that was inherited by the parent subtype from the grandparent subtype, but only if the parent subtype statically matches the first subtype of the parent type. An inherited representation aspect is overridden by a subsequent aspect_specification or representation item that specifies a different value for the same aspect of the type or subtype. 15/3

In contrast, whether type-related operational aspects are inherited by a derived type depends on each specific aspect; unless specified, an operational aspect is not inherited. When type-related operational aspects are inherited by a derived type, aspects that were directly specified by aspect_specifications or operational items that are visible at any point within the immediate scope of the derived type declaration, or (in the case where the parent is derived) that were inherited by the parent type from the grandparent type, are inherited. An inherited operational aspect is overridden by an aspect_specification or operational item that specifies the same aspect of the type. 15.1/5

When a type-related operational aspect is inherited, the rules for inheritance depend on the nature of the aspect (see 13.1.1). Unless otherwise specified for a given aspect, these rules are as follows: 15.2/5

- For an operational aspect that is a value, the inherited aspect has the same value; 15.3/5
- For an operational aspect that is a name: 15.4/5
  - if the name denotes one or more primitive subprograms of the type, the inherited aspect is a name that denotes the corresponding primitive subprogram(s) of the derived type; 15.5/5

15.6/5 - otherwise, the inherited aspect is a name that denotes the same entity or entities as the original aspect;

15.7/5 - For an operational aspect that is an identifier specific to the aspect, the inherited aspect is the same identifier;

15.8/5 - For an operational aspect that is an expression or an aggregate, the inherited aspect is a corresponding expression or aggregate where each name, value, and identifier follows these same rules for inheritance.

16 Each aspect of representation of an entity is as follows:

17 - If the aspect is *specified* for the entity, meaning that it is either directly specified or inherited, then that aspect of the entity is as specified, except in the case of Storage_Size, which specifies a minimum.

18 - If an aspect of representation of an entity is not specified, it is chosen by default in an unspecified manner.

18.1/5 If an operational aspect is *specified* for an entity (meaning that it is either directly specified or, if type-related or subtype-specific, inherited), then that aspect of the entity is as specified. Otherwise, the aspect of the entity has the default value for that aspect. For aspects that are neither type-related nor subtype-specific, the terms “specified” and “directly specified” are equivalent.

18.2/5 An aspect_specification or representation item that specifies a representation aspect that would have been chosen in the absence of the aspect_specification or representation item is said to be *confirming*. The aspect value specified in this case is said to be a *confirming* representation aspect value. Other values of the aspect are said to be *nonconfirming*, as are the aspect_specifications and representation items that specified them. Similarly, an aspect_specification or operational item that specifies an operational aspect to be the same as the definition it would have by default is said to be *confirming*; otherwise it is *nonconfirming*.

Dynamic Semantics

19/1 For the elaboration of an aspect_clause, any evaluable constructs within it are evaluated.

Implementation Permissions

20/3 An implementation may interpret representation aspects in an implementation-defined manner. An implementation may place implementation-defined restrictions on the specification of representation aspects. A *recommended level of support* is defined for the specification of representation aspects and related features in each subclause. These recommendations are changed to requirements for implementations that support the Systems Programming Annex (see C.2, “Required Representation Support”).

Implementation Advice

21/3 The recommended level of support for the specification of all representation aspects is qualified as follows:

21.1/3 - A confirming specification for a representation aspect should be supported.

22/5 - An implementation is not required to support the specification for a representation aspect that contains nonstatic expressions, unless each nonstatic expression is a name that statically denotes a constant declared before the entity.

23/5 - An implementation is not required to support a specification for the Object_Size or Size for a given composite subtype, nor the size or storage place for an object (including a component) of a

given composite subtype, unless the constraints on the subtype and its composite subcomponents (if any) are all static constraints.

- An implementation is not required to support specifying a nonconfirming representation aspect value if it can cause an aliased object or an object of a by-reference type to be allocated at a nonaddressable location or, when the alignment attribute of the subtype of such an object is nonzero, at an address that is not an integral multiple of that alignment. 24/5
- An implementation is not required to support specifying a nonconfirming representation aspect value if it can cause an aliased object of an elementary type to have a size other than that which would have been chosen by default. 25/5
- An implementation is not required to support specifying a nonconfirming representation aspect value if it can cause an aliased object of a composite type, or an object whose type is by-reference, to have a size smaller than that which would have been chosen by default. 26/5
- An implementation is not required to support specifying a nonconfirming subtype-specific representation aspect value for an indefinite or abstract subtype. 27/5

For purposes of these rules, the determination of whether specifying a representation aspect value for a type *can cause* an object to have some property is based solely on the properties of the type itself, not on any available information about how the type is used. In particular, it presumes that minimally aligned objects of this type can be declared at some point. 28/5

NOTE Aspects that can be specified are defined throughout this document, and are summarized in K.1. 29/3

13.1.1 Aspect Specifications

Certain representation or operational aspects of an entity may be specified as part of its declaration using an aspect_specification, rather than using a separate representation or operational item. The declaration with the aspect_specification is termed the *associated declaration*. 1/3

Syntax

aspect_specification ::= 2/3
  with aspect_mark [=> aspect_definition] {,
       aspect_mark [=> aspect_definition] }

aspect_mark ::= *aspect*_identifier['Class] 3/3

aspect_definition ::= 4/5
   name | expression | identifier
 | aggregate | global_aspect_definition

Name Resolution Rules

An aspect_mark identifies an aspect of the entity defined by the associated declaration (the *associated entity*); the aspect denotes an object, a value, an expression, an aggregate, a subprogram, or some other kind of entity. If the aspect_mark identifies: 5/5

- an aspect that denotes an object, the aspect_definition shall be a name. The expected type for the name is the type of the identified aspect of the associated entity; 6/3
- an aspect that is a value or an expression, the aspect_definition shall be an expression. The expected type for the expression is the type of the identified aspect of the associated entity; 7/3
- an aspect that is an aggregate, the aspect definition shall be an expression that is an aggregate, with the form of the aggregate determined by the identified aspect; 7.1/5

8/6 - an aspect that denotes a subprogram, the aspect_definition shall be a name; the expected profile for the name is the profile required for the aspect of the associated entity; additionally, unless otherwise specified, only subprograms declared within the declarative region of the entity associated with the aspect are considered;

9/3 - an aspect that denotes some other kind of entity, the aspect_definition shall be a name, and the name shall resolve to denote an entity of the appropriate kind;

10/3 - an aspect that is given by an identifier specific to the aspect, the aspect_definition shall be an identifier, and the identifier shall be one of the identifiers specific to the identified aspect.

11/5 The usage names in an aspect_definition associated with a declaration are not resolved at the point of the associated declaration, but rather are resolved at the end of the immediately enclosing declaration list, or in the case of the declaration of a library unit, at the end of the visible part of the entity.

12/5 If the associated declaration is for a subprogram, entry, or access-to-subprogram type, the names of the formal parameters are directly visible within the aspect_definition, as are certain attributes, as specified elsewhere in this document for the identified aspect. If the associated declaration is a type_declaration, within the aspect_definition the names of any visible components, protected subprograms, and entries are directly visible, and the name of the first subtype denotes the current instance of the type (see 8.6). If the associated declaration is a subtype_declaration, within the aspect_definition the name of the new subtype denotes the current instance of the subtype.

Legality Rules

13/3 If the first freezing point of the associated entity comes before the end of the immediately enclosing declaration list, then each usage name in the aspect_definition shall resolve to the same entity at the first freezing point as it does at the end of the immediately enclosing declaration list.

13.1/6 If a name or expression within an aspect_specification causes freezing when the entity associated with the aspect_specification is frozen (see 13.14), that name or expression shall not cause freezing of this associated entity.

14/3 At most one occurrence of each aspect_mark is allowed within a single aspect_specification. The aspect identified by the aspect_mark shall be an aspect that can be specified for the associated entity (or view of the entity defined by the associated declaration).

15/3 The aspect_definition associated with a given aspect_mark may be omitted only when the aspect_mark identifies an aspect of a boolean type, in which case it is equivalent to the aspect_definition being specified as True.

16/3 If the aspect_mark includes 'Class, then the associated entity shall be a tagged type or a primitive subprogram of a tagged type.

17/5 Unless otherwise specified for a specific aspect, a language-defined aspect cannot be specified on a renaming_declaration or a generic_formal_parameter_declaration.

18/5 Unless specified otherwise, a language-defined aspect shall not be specified in an aspect_specification given on a completion of a program unit.

18.1/4 If an aspect of a derived type is inherited from an ancestor type and has the boolean value True, the inherited value shall not be overridden to have the value False for the derived type, unless otherwise specified in this document.

If a given aspect is type-related and inherited, then within an aspect_definition for the aspect, if a name resolves to denote multiple visible subprograms, all or none of the denoted subprograms shall be primitives of the associated type. 18.2/5

Certain type-related aspects are defined to be *nonoverridable*; all such aspects are inherited by derived types according to the rules given in 13.1. Any legality rule associated with a nonoverridable aspect is rechecked for the derived type, if the derived type is not abstract. Certain type-related and subtype-specific aspects are defined to be *additive*; such aspects are not inherited, but they can *apply* to the types derived from, or the subtypes based on, the original type or subtype, as defined for each such aspect. Finally, certain type-related aspects are *implicitly composed*; such aspects are not inherited, but rather a default implementation for a derived type is provided, as defined for each such aspect, based on that of its parent type, presuming the aspect for the parent type is available where the derived type is declared, plus those of any new components added as part of a type extension. 18.3/5

For a nonoverridable aspect of a type *T* that denotes one or more subprograms with a parameter or result of type *T* or **access** *T*, all of the denoted subprograms shall be primitive for *T*. The same restriction applies to elements that denote subprograms within a nonoverridable aspect that is in the form of an aggregate. 18.4/6

If a nonoverridable aspect is directly specified for a type *T*, then any explicit specification of that aspect for any descendant of *T* (other than *T* itself) shall be *confirming*. In the case of an aspect that is a name, this means that the specified name shall *match* the inherited aspect in the sense that it shall denote the same declarations as would the inherited name. Similarly, for an aspect that is an expression or an aggregate, confirming means the defining expression is fully conformant (see 6.3.1) with the defining expression for the inherited aspect, with the added rule that an identifier that is specific to the aspect is the same as the corresponding identifier in the inherited aspect. 18.5/6

If a full type has a partial view, and a given nonoverridable aspect is allowed for both the full view and the partial view, then the given aspect for the partial view and the full view shall be the same: the aspect shall be directly specified only on the partial view; if the full type inherits the aspect, then a matching definition shall be specified (directly or by inheritance) for the partial view. 18.6/6

If a type inherits a nonoverridable aspect from multiple ancestors, the value of the aspect inherited from any given ancestor shall be confirming of the values inherited from all other ancestors. 18.7/6

In addition to the places where Legality Rules normally apply (see 12.3), these rules about nonoverridable aspects also apply in the private part of an instance of a generic unit. 18.8/6

Static Semantics

Depending on which aspect is identified by the aspect_mark, an aspect_definition specifies: 19/3

- a name that denotes a subprogram, object, or other kind of entity; 20/3
- an expression (other than an aggregate), which is either evaluated to produce a single value, or which (as in a precondition) is to be evaluated at particular points during later execution; 21/5
- an identifier specific to the aspect; or 22/5
- an aggregate, which is positional or named, and is composed of elements of any of these four kinds of constructs. 22.1/5

The identified aspect of the associated entity, or in some cases, the view of the entity defined by the declaration, is as specified by the aspect_definition (or by the default of True when boolean). Whether an aspect_specification *applies* to an entity or only to the particular view of the entity defined by the declaration is determined by the aspect_mark and the kind of entity. The following aspects are view specific: 23/3

24/3 • An aspect specified on an object_declaration;

25/3 • An aspect specified on a subprogram_declaration;

26/3 • An aspect specified on a renaming_declaration.

27/3 All other aspect_specifications are associated with the entity, and *apply* to all views of the entity, unless otherwise specified in this document.

28/4 If the aspect_mark includes 'Class (a *class-wide aspect*), then, unless specified otherwise for a particular class-wide aspect:

29/3 • if the associated entity is a tagged type, the specification *applies* to all descendants of the type;

30/3 • if the associated entity is a primitive subprogram of a tagged type *T*, the specification *applies* to the corresponding primitive subprogram of all descendants of *T*.

31/3 All specifiable operational and representation attributes may be specified with an aspect_specification instead of an attribute_definition_clause (see 13.3).

32/6 Some aspects are defined to be *library unit aspects*. Library unit aspects are of type Boolean. The expression specifying a library unit aspect shall be static. Library unit aspects are defined for all program units, but shall be specified only for library units that are not library_unit_renaming_declarations. Notwithstanding what this document says elsewhere, the expression of a library unit aspect is resolved and evaluated at the point where it occurs in the aspect_specification, rather than the first freezing point of the associated unit.

33/3 In addition, other operational and representation aspects not associated with specifiable attributes or representation pragmas may be specified, as specified elsewhere in this document.

34/4 *This paragraph was deleted.*

35/3 If a Legality Rule or Static Semantics rule only applies when a particular aspect has been specified, the aspect is considered to have been specified only when the aspect_specification or attribute_definition_clause is visible (see 8.3) at the point of the application of the rule.

36/3 Alternative legality and semantics rules may apply for particular aspects, as specified elsewhere in this document.

Dynamic Semantics

37/5 At the freezing point of the associated entity, the aspect_specification is elaborated. When appearing in a construct other than a declaration, an aspect_specification is elaborated as part of the execution of the construct. The elaboration of the aspect_specification consists of the elaboration of each aspect_definition in an arbitrary order. The elaboration of an aspect_definition includes the evaluation of any name or expression that is part of the aspect_definition unless the part is itself an expression. If the corresponding aspect (or part thereof) represents an expression (as in a precondition), the elaboration of that part has no effect; the expression is evaluated later at points within the execution as specified elsewhere in this document for the particular aspect.

Implementation Permissions

38/3 Implementations may support implementation-defined aspects. The aspect_specification for an implementation-defined aspect may use an implementation-defined syntax for the aspect_definition, and may follow implementation-defined legality and semantics rules.

39/5 An implementation may ignore the specification of an unrecognized aspect; if an implementation chooses to ignore such an aspect specification (as opposed to rejecting it), then it has no effect on the semantics of

the program except for possibly (and this is not required) the rejection of syntax errors within the aspect_definition.

13.2 Packed Types

The Pack aspect having the value True specifies that storage minimization should be the main criterion when selecting the representation of a composite type. 1/3

Paragraphs 2 through 4 were moved to Annex J, "Obsolescent Features".

Static Semantics

For a full type declaration of a composite type, the following language-defined representation aspect may be specified: 5/3

Pack — The type of aspect Pack is Boolean. When aspect Pack is True for a type, the type (or the extension part) is said to be *packed*. For a type extension, the parent part is packed as for the parent type, and specifying Pack causes packing only of the extension part. 5.1/3

If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False. 5.2/3

Implementation Advice

If a type is packed, then the implementation should try to minimize storage allocated to objects of the type, possibly at the expense of speed of accessing components, subject to reasonable complexity in addressing calculations. 6

This paragraph was deleted. 6.1/4

The recommended level of support for the Pack aspect is: 7/3

- Any component of a packed type that is of a by-reference type, that is specified as independently addressable, or that contains an aliased part, shall be aligned according to the alignment of its subtype. 7.1/4
- For a packed record type, the components should be packed as tightly as possible subject to the above alignment requirements, the Sizes of the component subtypes, and any record_representation_clause that applies to the type; the implementation is allowed to reorder components or cross aligned word boundaries to improve the packing. A component whose Size is greater than the word size may be allocated an integral number of words. 8/5
- For a packed array type, if the Size of the component subtype is less than or equal to the word size, Component_Size should be less than or equal to the Size of the component subtype, rounded up to the nearest factor of the word size, unless this would violate the above alignment requirements. 9/4

13.3 Operational and Representation Attributes

The values of certain implementation-dependent characteristics can be obtained by interrogating appropriate operational or representation attributes. Some of these attributes are specifiable via an attribute_definition_clause. 1/1

Syntax

2 attribute_definition_clause ::=
for local_name'attribute_designator **use** expression;
| **for** local_name'attribute_designator **use** name;

Name Resolution Rules

3 For an attribute_definition_clause that specifies an attribute that denotes a value, the form with an expression shall be used. Otherwise, the form with a name shall be used.

4 For an attribute_definition_clause that specifies an attribute that denotes a value or an object, the expected type for the expression or name is that of the attribute. For an attribute_definition_clause that specifies an attribute that denotes a subprogram, the expected profile for the name is the profile required for the attribute. For an attribute_definition_clause that specifies an attribute that denotes some other kind of entity, the name shall resolve to denote an entity of the appropriate kind.

Legality Rules

5/3 An attribute_designator is allowed in an attribute_definition_clause only if this Reference Manual explicitly allows it, or for an implementation-defined attribute if the implementation allows it. Each specifiable attribute constitutes an operational aspect or aspect of representation; the name of the aspect is that of the attribute.

6/5 *This paragraph was deleted.*

Static Semantics

7/2 A *Size clause* is an attribute_definition_clause whose attribute_designator is Size. Similar definitions apply to the other specifiable attributes.

8 A *storage element* is an addressable element of storage in the machine. A *word* is the largest amount of storage that can be conveniently and efficiently manipulated by the hardware, given the implementation's run-time model. A word consists of an integral number of storage elements.

8.1/3 A *machine scalar* is an amount of storage that can be conveniently and efficiently loaded, stored, or operated upon by the hardware. Machine scalars consist of an integral number of storage elements. The set of machine scalars is implementation defined, but includes at least the storage element and the word. Machine scalars are used to interpret component_clauses when the nondefault bit ordering applies.

9/5 The following representation attributes are defined: Address, Alignment, Size, Object_Size, Storage_Size, Component_Size, Has_Same_Storage, and Overlaps_Storage.

10/1 For a prefix X that denotes an object, program unit, or label:

11 X'Address Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address.

11.1/3 The prefix of X'Address shall not statically denote a subprogram that has convention Intrinsic. X'Address raises Program_Error if X denotes a subprogram that has convention Intrinsic.

12 Address may be specified for stand-alone objects and for program units via an attribute_definition_clause.

Erroneous Execution

If an Address is specified, it is the programmer's responsibility to ensure that the address is valid and appropriate for the entity and its use; otherwise, program execution is erroneous. 13/3

Implementation Advice

For an array X, X'Address should point at the first component of the array, and not at the array bounds. 14

The recommended level of support for the Address attribute is: 15

- X'Address should produce a useful result if X is an object that is aliased or of a by-reference type, or is an entity whose Address has been specified. 16
- An implementation should support Address clauses for imported subprograms. 17
- *This paragraph was deleted.* 18/2
- If the Address of an object is specified, or it is imported or exported, then the implementation should not perform optimizations based on assumptions of no aliases. 19

NOTE 1 The specification of a link name with the Link_Name aspect (see B.1) for a subprogram or object is an alternative to explicit specification of its link-time address, allowing a link-time directive to place the subprogram or object within memory. 20

NOTE 2 The rules for the Size attribute imply, for an aliased object X, that if X'Size = Storage_Unit, then X'Address points at a storage element containing all of the bits of X, and only the bits of X. 21

Static Semantics

For a prefix X that denotes an object: 22/2

X'Alignment The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero). 23/2

This paragraph was deleted. 24/2

Alignment may be specified for stand-alone objects via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 25/2

This paragraph was deleted. 26/2

For every subtype S: 26.1/2

S'Alignment The value of this attribute is of type *universal_integer*, and nonnegative. 26.2/2

For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item. 26.3/2

Alignment may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 26.4/2

Erroneous Execution

Program execution is erroneous if an Address clause is given that conflicts with the Alignment. 27

For an object that is not allocated under control of the implementation, execution is erroneous if the object is not aligned according to its Alignment. 28/2

Implementation Advice

For any tagged specific subtype *S*, *S*'Class'Alignment should equal *S*'Alignment. 28.1/3

The recommended level of support for the Alignment attribute for subtypes is: 29

30/2 • An implementation should support an Alignment clause for a discrete type, fixed point type, record type, or array type, specifying an Alignment value that is zero or a power of two, subject to the following:

31/5 • An implementation is not required to support an Alignment clause for a signed integer type specifying an Alignment greater than the largest Alignment value that is ever chosen by default by the implementation for any signed integer type. A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types.

32/5 • An implementation is not required to support a nonconfirming Alignment clause that can cause the creation of an object of an elementary type that cannot be easily loaded and stored by available machine instructions.

32.1/5 • An implementation is not required to support an Alignment specified for a derived tagged type that is not a multiple of the Alignment of the parent type. An implementation is not required to support a nonconfirming Alignment specified for a derived untagged by-reference type.

33 The recommended level of support for the Alignment attribute for objects is:

34/2 • *This paragraph was deleted.*

35 • For stand-alone library-level objects of statically constrained subtypes, the implementation should support all Alignments supported by the target linker. For example, page alignment is likely to be supported for such objects, but not for subtypes.

35.1/2 • For other objects, an implementation should at least support the alignments supported for their subtype, subject to the following:

35.2/5 • An implementation is not required to support Alignments specified for objects of a by-reference type or for objects of types containing aliased subcomponents if the specified Alignment is not a multiple of the Alignment of the subtype of the object.

36 NOTE 3 Alignment is a subtype-specific attribute.

37/2 *This paragraph was deleted.*

38/3 NOTE 4 A component_clause, Component_Size clause, or specifying the Pack aspect as True can override a specified Alignment.

Static Semantics

39/1 For a prefix X that denotes an object:

40 X'Size Denotes the size in bits of the representation of the object. The value of this attribute is of the type *universal_integer*.

41 Size may be specified for stand-alone objects via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative.

Implementation Advice

41.1/2 The size of an array object should not include its bounds.

42/5 The recommended level of support for the Size attribute of objects is the same as for subtypes (see below), except that only a confirming Size clause is required to be supported for an aliased elementary object.

43/2 • *This paragraph was deleted.*

Static Semantics

44 For every subtype S:

45 S'Size If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S:

- A record component of subtype S when the record type is packed. 46
- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype. 47

If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type *universal_integer*. The Size of an object is at least as large as that of its subtype, unless the object's Size is determined by a Size clause, a component_clause, or a Component_Size clause. Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative. 48

Implementation Requirements

In an implementation, Boolean'Size shall be 1. 49

Implementation Advice

If the Size of a subtype is nonconfirming and allows for efficient independent addressability (see 9.10) on the target architecture, then the Object_Size of the subtype should have the same value in the absence of an explicit specification of a different value. 50/5

Paragraphs 51 and 52 were moved to the Implementation Advice for attribute Object_Size.

A Size clause on a composite subtype should not affect the internal layout of components. 53

The recommended level of support for the Size attribute of subtypes is: 54

- The Size (if not specified) of a static discrete or fixed point subtype should be the number of bits necessary to represent each value belonging to the subtype using an unbiased representation, leaving space for a sign bit only if the subtype contains negative values. If such a subtype is a first subtype, then an implementation should support a specified Size for it that reflects this representation. 55/5
- For a subtype implemented with levels of indirection, the Size should include the size of the pointers, but not the size of what they point at. 56
- An implementation should support a Size clause for a discrete type, fixed point type, record type, or array type, subject to the following: 56.1/2
  - An implementation is not required to support a Size clause for a signed integer type specifying a Size greater than that of the largest signed integer type supported by the implementation in the absence of a size clause (that is, when the size is chosen by default). A corresponding limitation may be imposed for modular integer types, fixed point types, enumeration types, record types, and array types. 56.2/5
  - A nonconfirming size clause for the first subtype of a derived untagged by-reference type is not required to be supported. 56.3/5

NOTE 5 Size is a subtype-specific attribute. 57

NOTE 6 A component_clause or Component_Size clause can override a specified Size. Aspect Pack cannot. 58/3

Static Semantics

For every subtype S: 58.1/5

S'Object_Size 58.2/5

If S is definite, denotes the size (in bits) of a stand-alone aliased object, or a component of subtype S in the absence of an aspect_specification or representation item that specifies the size of the object or component. If S is indefinite, the meaning is implementation-defined. The value of this attribute is of the type *universal_integer*. If not specified otherwise, the Object_Size of a subtype is at least as large as the Size of the subtype. Object_Size may be

specified via an attribute_definition_clause; the expression of such a clause shall be static and its value nonnegative. All aliased objects with nominal subtype S have the size S'Object_Size. In the absence of an explicit specification, the Object_Size of a subtype S defined by a subtype_indication without a constraint, is that of the value of the Object_Size of the subtype denoted by the subtype_mark of the subtype_indication, at the point of this definition.

Implementation Advice

58.3/5 If S is a definite first subtype and S'Object_Size is not specified, S'Object_Size should be the smallest multiple of the storage element size larger than or equal to S'Size that is consistent with the alignment of S.

58.4/5 If X denotes an object (including a component) of subtype S, X'Size should equal S'Object_Size, unless:

58.5/5 • X'Size is specified; or

58.6/5 • X is a nonaliased stand-alone object; or

58.7/5 • The size of X is determined by a component_clause or Component_Size clause; or

58.8/5 • The type containing component X is packed.

58.9/5 An Object_Size clause on a composite type should not affect the internal layout of components.

58.10/5 The recommended level of support for the Object_Size attribute of subtypes is:

58.11/5 • If S is a static signed integer subtype, the implementation should support the specification of S'Object_Size to match the size of any signed integer base subtype provided by the implementation that is no smaller than S'Size. Corresponding support is expected for modular integer subtypes, fixed point subtypes, and enumeration subtypes.

58.12/5 • If S is an array or record subtype with static constraints and S is not a first subtype of a derived untagged by-reference type, the implementation should support the specification of S'Object_Size to be any multiple of the storage element size that is consistent with the alignment of S, that is no smaller than S'Size, and that is no larger than that of the largest composite subtype supported by the implementation.

58.13/5 • If S is some other subtype, only confirming specifications of Object_Size are required to be supported.

Static Semantics

59/1 For a prefix T that denotes a task object (after any implicit dereference):

60/3 T'Storage_Size

Denotes the number of storage elements reserved for the task. The value of this attribute is of the type *universal_integer*. The Storage_Size includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the “task control block” used by some implementations.) If the aspect Storage_Size is specified for the type of the object, the value of the Storage_Size attribute is at least the value determined by the aspect.

61/3 Aspect Storage_Size specifies the amount of storage to be reserved for the execution of a task.

Paragraphs 62 through 65 were moved to Annex J, “Obsolescent Features”.

Static Semantics

65.1/6 For a task type (including the anonymous type of a single_task_declaration) or a subprogram, the following language-defined representation aspect may be specified:

65.2/3 Storage_Size The Storage_Size aspect is an expression, which shall be of any integer type.

Legality Rules

The Storage_Size aspect shall not be specified for a task interface type. 65.3/3

If the Storage_Size aspect is specified for a subprogram, the expression shall be static. 65.4/6

Dynamic Semantics

When a task object is created, the expression specified for the Storage_Size aspect of its type is evaluated; the Storage_Size attribute of the newly created task object is at least the value of the expression. For the environment task, the expression specified for the Storage_Size aspect of the main subprogram (if any) is evaluated instead. 66/6

The Storage_Size aspect has no effect if it is specified for a subprogram other than the main subprogram; the Storage_Size expression is not associated with any task. 66.1/6

At the point of task object creation, or upon task activation, Storage_Error is raised if there is insufficient free storage to accommodate the requested Storage_Size. 67

Static Semantics

For a prefix X that denotes an array subtype or array object (after any implicit dereference): 68/1

X'Component_Size 69

Denotes the size in bits of components of the type of X. The value of this attribute is of type *universal_integer*.

Component_Size may be specified for array types via an attribute_definition_clause; the expression of such a clause shall be static, and its value nonnegative. 70

Implementation Advice

The recommended level of support for the Component_Size attribute is: 71

- An implementation is not required to support specified Component_Sizes that are less than the Size of the component subtype. 72/5
- An implementation should support specified Component_Sizes that are factors and multiples of the word size. For such Component_Sizes, the array should contain no gaps between components. For other Component_Sizes (if supported), the array should contain no gaps between components when Pack is also specified; the implementation should forbid this combination in cases where it cannot support a no-gaps representation. 73/3

Static Semantics

For a prefix X that denotes an object: 73.1/3

X'Has_Same_Storage 73.2/4

X'Has_Same_Storage denotes a function with the following specification:

```
function X'Has_Same_Storage (Arg : any_type)
  return Boolean
```
73.3/3

The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved. It returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X and the objects occupy at least one bit; otherwise, it returns False. 73.4/4

For a prefix X that denotes an object: 73.5/3

X'Overlaps_Storage 73.6/3

X'Overlaps_Storage denotes a function with the following specification:

73.7/3
```
function X'Overlaps_Storage (Arg : any_type)
  return Boolean
```

73.8/3 The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved and returns True if the representation of the object denoted by the actual parameter shares at least one bit with the representation of the object denoted by X; otherwise, it returns False.

73.9/3 NOTE 7 X'Has_Same_Storage(Y) implies X'Overlaps_Storage(Y).

73.10/3 NOTE 8 X'Has_Same_Storage(Y) and X'Overlaps_Storage(Y) are not considered to be reads of X and Y.

Static Semantics

73.11/3 The following type-related operational attribute is defined: External_Tag.

74/1 For every subtype S of a tagged type *T* (specific or class-wide):

75/3 S'External_Tag

S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 13.13.2. The value of External_Tag is never inherited; the default value is always used unless a new value is directly specified for a type.

Dynamic Semantics

75.1/6 If two tagged types X and Y both exist in a given partition, X and Y are declared by distinct declarations, and a user-specified X'External_Tag is the same as Y'External_Tag, then Program_Error is raised at one or more of the following places:

75.2/6 - an attribute_definition_clause for X'External_Tag;
75.3/6 - an attribute_definition_clause for Y'External_Tag;
75.4/6 - the freezing point of X;
75.5/6 - the freezing point of Y.

Implementation Requirements

76/6 In an implementation, the default external tag for each specific tagged type declared in a partition shall be distinct, so long as the type is declared by a distinct declaration and is outside an instance of a generic body. If the compilation unit in which a given tagged type is declared, and all compilation units on which it semantically depends, are the same in two different partitions, then the external tag for the type shall be the same in the two partitions. What it means for a compilation unit to be the same in two different partitions is implementation defined. At a minimum, if the compilation unit is not recompiled between building the two different partitions that include it, the compilation unit is considered the same in the two partitions.

Implementation Permissions

76.1/6 If a user-specified external tag X'External_Tag is the same as Y'External_Tag for some other tagged type declared by a different declaration in the partition, an implementation may disallow the partition, or raise Program_Error upon the elaboration of either type.

77/2 NOTE 9 The following language-defined attributes are specifiable, at least for some of the kinds of entities to which they apply: Address, Alignment, Bit_Order, Component_Size, External_Tag, Input, Machine_Radix, Output, Read, Size, Small, Storage_Pool, Storage_Size, Stream_Size, and Write.

NOTE 10 It follows from the general rules in 13.1 that if one writes "**for** X'Size **use** Y;" then the X'Size attribute_reference will return Y (assuming the implementation allows the Size clause). The same is true for all of the specifiable attributes except Storage_Size. 78

Examples

Examples of attribute definition clauses: 79

```
Byte : constant := 8;                                                    80
Page : constant := 2**12;

type Medium is range 0 .. 65_000;                                        81
for Medium'Size use 2*Byte;
for Medium'Alignment use 2;
Device_Register : Medium;
for Device_Register'Size use Medium'Size;
for Device_Register'Address use
   System.Storage_Elements.To_Address(16#FFFF_0020#);

type Short is delta 0.01 range -100.0 .. 100.0;                          82
for Short'Size use 15;

for Car_Name'Storage_Size use -- specify access type's storage pool size 83
        2000*((Car'Size/System.Storage_Unit) +1); -- approximately 2000 cars

function My_Input(Stream : not null access                               84/2
Ada.Streams.Root_Stream_Type'Class)
   return T;
for T'Input use My_Input; -- see 13.13.2
```

In the Size clause for Short, fifteen bits is the minimum necessary, since the type definition requires Short'Small <= 2**(–7). 85/5

13.4 Enumeration Representation Clauses

An enumeration_representation_clause specifies the internal codes for enumeration literals. 1

Syntax

enumeration_representation_clause ::= 2
   for *first_subtype*_local_name **use** enumeration_aggregate;

enumeration_aggregate ::= array_aggregate 3

Name Resolution Rules

The enumeration_aggregate shall be written as a one-dimensional array_aggregate, for which the index subtype is the unconstrained subtype of the enumeration type, and each component expression is expected to be of any integer type. 4

Legality Rules

The *first_subtype*_local_name of an enumeration_representation_clause shall denote an enumeration subtype. 5

Each component of the array_aggregate shall be given by an expression rather than a <>. The expressions given in the array_aggregate shall be static, and shall specify distinct integer codes for each value of the enumeration type; the associated integer codes shall satisfy the predefined ordering relation of the type. 6/2

Static Semantics

7 An enumeration_representation_clause specifies the *coding* aspect of representation. The coding consists of the *internal code* for each enumeration literal, that is, the integral value used internally to represent each literal.

Implementation Requirements

8 For nonboolean enumeration types, if the coding is not specified for the type, then for each value of the type, the internal code shall be equal to its position number.

Implementation Advice

9 The recommended level of support for enumeration_representation_clauses is:

10/5 • An implementation should support at least the internal codes in the range System.Min_Int .. System.Max_Int. An implementation is not required to support enumeration_representation_-clauses for boolean types.

Static Semantics

10.1/5 For every discrete subtype S, the following attributes are defined:

10.2/5 S'Enum_Rep S'Enum_Rep denotes a function with the following specification:

10.3/5
```
function S'Enum_Rep (Arg : S'Base) return universal_integer
```

10.4/5 This function returns the representation value of the value of Arg, as a value of type *universal_integer*. The *representation value* is the internal code specified in an enumeration representation clause, if any, for the type corresponding to the value of Arg, and otherwise is the position number of the value.

10.5/5 S'Enum_Val S'Enum_Val denotes a function with the following specification:

10.6/5
```
function S'Enum_Val (Arg : universal_integer) return S'Base
```

10.7/5 This function returns a value of the type of S whose representation value equals the value of Arg. For the evaluation of a call on S'Enum_Val, if there is no value in the base range of its type with the given representation value, Constraint_Error is raised.

11/5 NOTE Attribute Enum_Rep can be used to query the internal codes used for an enumeration type; attribute Enum_Val can be used to convert from an internal code to an enumeration value. The other attributes of the type, such as Succ, Pred, and Pos, are unaffected by an enumeration_representation_clause. For example, Pos always returns the position number, *not* an internal integer code that was specified in an enumeration_representation_clause.

Examples

12/5 *Examples of enumeration representation clauses:*

13
```
type Mix_Code is (ADD, SUB, MUL, LDA, STA, STZ);
```

14
```
for Mix_Code use
   (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24, STZ =>33);
```

15/5
```
-- See 3.5.2.
for Roman_Digit use ('I' => 1,
                     'V' => 5,
                     'X' => 10,
                     'L' => 50,
                     'C' => 100,
                     'D' => 500,
                     'M' => 1000);
```

16/5 *-- For an example of the use of attribute Enum_Rep, see 4.2.1.*

13.5 Record Layout

The *(record) layout* aspect of representation consists of the *storage places* for some or all components, that is, storage place attributes of the components. The layout can be specified with a record_representation_clause. 1

13.5.1 Record Representation Clauses

A record_representation_clause specifies the storage representation of records and record extensions, that is, the order, position, and size of components (including discriminants, if any). 1

Syntax

record_representation_clause ::= 2/5
   for *first_subtype*_local_name **use**
      record [mod_clause]
         {component_clause}
      end record [local_name];

component_clause ::= 3
   *component*_local_name **at** position **range** first_bit .. last_bit;

position ::= *static*_expression 4

first_bit ::= *static*_simple_expression 5

last_bit ::= *static*_simple_expression 6

If a local_name appears at the end of the record_representation_clause, it shall repeat the *first_subtype*_local_name. 6.1/5

Name Resolution Rules

Each position, first_bit, and last_bit is expected to be of any integer type. 7

Legality Rules

The *first_subtype*_local_name of a record_representation_clause shall denote a specific record or record extension subtype. 8/2

If the *component*_local_name is a direct_name, the local_name shall denote a component of the type. For a record extension, the component shall not be inherited, and shall not be a discriminant that corresponds to a discriminant of the parent type. If the *component*_local_name has an attribute_designator, the direct_name of the local_name shall denote either the declaration of the type or a component of the type, and the attribute_designator shall denote an implementation-defined implicit component of the type. 9

The position, first_bit, and last_bit shall be static expressions. The value of position and first_bit shall be nonnegative. The value of last_bit shall be no less than first_bit – 1. 10

If the nondefault bit ordering applies to the type, then either: 10.1/2

- the value of last_bit shall be less than the size of the largest machine scalar; or 10.2/2
- the value of first_bit shall be zero and the value of last_bit + 1 shall be a multiple of System.Storage_Unit. 10.3/2

11 At most one component_clause is allowed for each component of the type, including for each discriminant (component_clauses may be given for some, all, or none of the components). Storage places within a component_list shall not overlap, unless they are for components in distinct variants of the same variant_part.

12 A name that denotes a component of a type is not allowed within a record_representation_clause for the type, except as the *component_*local_name of a component_clause.

Static Semantics

13/2 A record_representation_clause (without the mod_clause) specifies the layout.

13.1/2 If the default bit ordering applies to the type, the position, first_bit, and last_bit of each component_clause directly specify the position and size of the corresponding component.

13.2/3 If the nondefault bit ordering applies to the type, then the layout is determined as follows:

13.3/2 - the component_clauses for which the value of last_bit is greater than or equal to the size of the largest machine scalar directly specify the position and size of the corresponding component;

13.4/2 - for other component_clauses, all of the components having the same value of position are considered to be part of a single machine scalar, located at that position; this machine scalar has a size which is the smallest machine scalar size larger than the largest last_bit for all component_clauses at that position; the first_bit and last_bit of each component_clause are then interpreted as bit offsets in this machine scalar.

14 A record_representation_clause for a record extension does not override the layout of the parent part; if the layout was specified for the parent type, it is inherited by the record extension.

Implementation Permissions

15 An implementation may generate implementation-defined components (for example, one containing the offset of another component). An implementation may generate names that denote such implementation-defined components; such names shall be implementation-defined attribute_references. An implementation may allow such implementation-defined names to be used in record_representation_-clauses. An implementation can restrict such component_clauses in any manner it sees fit.

16 If a record_representation_clause is given for an untagged derived type, the storage place attributes for all of the components of the derived type may differ from those of the corresponding components of the parent type, even for components whose storage place is not specified explicitly in the record_-representation_clause.

Implementation Advice

17 The recommended level of support for record_representation_clauses is:

17.1/2 - An implementation should support machine scalars that correspond to all of the integer, floating point, and address formats supported by the machine.

18 - An implementation should support storage places that can be extracted with a load, mask, shift sequence of machine code, and set with a load, shift, mask, store sequence, given the available machine instructions and run-time model.

19 - A storage place should be supported if its size is equal to the Size of the component subtype, and it starts and ends on a boundary that obeys the Alignment of the component subtype.

20/2 - For a component with a subtype whose Size is less than the word size, any storage place that does not cross an aligned word boundary should be supported.

- An implementation may reserve a storage place for the tag field of a tagged type, and disallow other components from overlapping that place. 21
- An implementation is not required to support a component_clause for a component of an extension part if the storage place is not after the storage places of all components of the parent type, whether or not those storage places had been specified. 22/5

NOTE If no component_clause is given for a component, then the choice of the storage place for the component is left to the implementation. If component_clauses are given for all components, the record_representation_clause completely specifies the representation of the type and will be obeyed exactly by the implementation. 23

Examples

Example of specifying the layout of a record type: 24

```
Word : constant := 4;  -- storage element is byte, 4 bytes per word
```
25

```
type State         is (A,M,W,P);
type Mode          is (Fix, Dec, Exp, Signif);
```
26

```
type Byte_Mask     is array (0..7)  of Boolean with Component_Size => 1;
type State_Mask    is array (State) of Boolean with Component_Size => 1;
type Mode_Mask     is array (Mode)  of Boolean with Component_Size => 1;
```
27/5

```
type Program_Status_Word is
  record
      System_Mask        : Byte_Mask;
      Protection_Key     : Integer range 0 .. 3;
      Machine_State      : State_Mask;
      Interrupt_Cause    : Interruption_Code;
      Ilc                : Integer range 0 .. 3;
      Cc                 : Integer range 0 .. 3;
      Program_Mask       : Mode_Mask;
      Inst_Address       : Address;
end record;
```
28

```
for Program_Status_Word use
  record
      System_Mask      at 0*Word range 0  .. 7;
      Protection_Key   at 0*Word range 10 .. 11; -- bits 8,9 unused
      Machine_State    at 0*Word range 12 .. 15;
      Interrupt_Cause  at 0*Word range 16 .. 31;
      Ilc              at 1*Word range 0  .. 1;  -- second word
      Cc               at 1*Word range 2  .. 3;
      Program_Mask     at 1*Word range 4  .. 7;
      Inst_Address     at 1*Word range 8  .. 31;
  end record;
```
29

```
for Program_Status_Word'Size use 8*System.Storage_Unit;
for Program_Status_Word'Alignment use 8;
```
30

The record_representation_clause defines the record layout. The Size clause guarantees that (at least) eight storage elements are used for objects of the type. The Alignment clause guarantees that aliased, imported, or exported objects of the type will have addresses divisible by eight. 31/5

13.5.2 Storage Place Attributes

Static Semantics

For a component C of a composite, non-array object R, the *storage place attributes* are defined: 1

R.C'Position If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*. 2/2

3/2 R.C'First_Bit

If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the first_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*.

4/2 R.C'Last_Bit

If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the last_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer*.

Implementation Advice

5 If a component is represented using some form of pointer (such as an offset) to the actual data of the component, and this data is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data, not the pointer. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes.

13.5.3 Bit Ordering

1 The Bit_Order attribute specifies the interpretation of the storage place attributes.

Static Semantics

2 A bit ordering is a method of interpreting the meaning of the storage place attributes. High_Order_First (known in the vernacular as “big endian”) means that the first bit of a storage element (bit 0) is the most significant bit (interpreting the sequence of bits that represent a component as an unsigned integer value). Low_Order_First (known in the vernacular as “little endian”) means the opposite: the first bit is the least significant.

3/5 For every specific record subtype S, the following representation attribute is defined:

4 S'Bit_Order Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. Bit_Order may be specified for specific record types via an attribute_definition_clause; the expression of such a clause shall be static.

5 If Word_Size = Storage_Unit, the default bit ordering is implementation defined. If Word_Size > Storage_Unit, the default bit ordering is the same as the ordering of storage elements in a word, when interpreted as an integer.

6 The storage place attributes of a component of a type are interpreted according to the bit ordering of the type.

Implementation Advice

7 The recommended level of support for the nondefault bit ordering is:

8/2 - The implementation should support the nondefault bit ordering in addition to the default bit ordering.

9/2 NOTE Bit_Order clauses make it possible to write record_representation_clauses that can be ported between machines having different bit ordering. They do not guarantee transparent exchange of data between such machines.

13.6 Change of Representation

A type_conversion (see 4.6) can be used to convert between two different representations of the same array or record. To convert an array from one representation to another, two array types with matching component subtypes and convertible index types are required. If one type has Pack specified and the other does not, then explicit conversion can be used to pack or unpack an array. 1/5

To convert an untagged record from one representation to another, two record types with a common ancestor type are required. Distinct representations can then be specified for the record types, and explicit conversion between the types can be used to effect a change in representation. 2/5

Examples

Example of change of representation: 3

```
-- Packed_Descriptor and Descriptor are two different types                    4
-- with identical characteristics, apart from their
-- representation
type Descriptor is                                                            5
    record
      -- components of a descriptor
    end record;
type Packed_Descriptor is new Descriptor;                                     6
for Packed_Descriptor use                                                     7
    record
      -- component clauses for some or for all components
    end record;
-- Change of representation can now be accomplished by explicit type conversions:  8
D : Descriptor;                                                               9
P : Packed_Descriptor;
P := Packed_Descriptor(D);   -- pack D                                        10
D := Descriptor(P);          -- unpack P
```

13.7 The Package System

For each implementation there is a library package called System which includes the definitions of certain configuration-dependent characteristics. 1

Static Semantics

The following language-defined library package exists: 2

```
package System                                                                3/5
   with Pure is
   type Name is implementation-defined-enumeration-type;                      4
   System_Name : constant Name := implementation-defined;
   -- System-Dependent Named Numbers:                                         5
   Min_Int               : constant := root_integer'First;                    6
   Max_Int               : constant := root_integer'Last;
   Max_Binary_Modulus    : constant := implementation-defined;                7
   Max_Nonbinary_Modulus : constant := implementation-defined;
   Max_Base_Digits       : constant := root_real'Digits;                      8
   Max_Digits            : constant := implementation-defined;
   Max_Mantissa          : constant := implementation-defined;                9
   Fine_Delta            : constant := implementation-defined;
```

10
```
   Tick                    : constant := implementation-defined;
```

11
```
   -- Storage-related Declarations:
```

12
```
   type Address is implementation-defined;
   Null_Address : constant Address;
```

13
```
   Storage_Unit : constant := implementation-defined;
   Word_Size    : constant := implementation-defined * Storage_Unit;
   Memory_Size  : constant := implementation-defined;
```

14/3
```
   -- Address Comparison:
   function "<" (Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function "<="(Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function ">" (Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function ">="(Left, Right : Address) return Boolean
      with Convention => Intrinsic;
   function "=" (Left, Right : Address) return Boolean
      with Convention => Intrinsic;
-- function "/=" (Left, Right : Address) return Boolean;
   -- "/=" is implicitly defined
```

15/2
```
   -- Other System-Dependent Declarations:
   type Bit_Order is (High_Order_First, Low_Order_First);
   Default_Bit_Order : constant Bit_Order := implementation-defined;
```

16
```
   -- Priority-related declarations (see D.1):
   subtype Any_Priority is Integer range implementation-defined;
   subtype Priority is Any_Priority range Any_Priority'First ..
             implementation-defined;
   subtype Interrupt_Priority is Any_Priority range Priority'Last+1 ..
             Any_Priority'Last;
```

17
```
   Default_Priority : constant Priority :=
             (Priority'First + Priority'Last)/2;
```

18
```
private
   ... -- not specified by the language
end System;
```

19 Name is an enumeration subtype. Values of type Name are the names of alternative machine configurations handled by the implementation. System_Name represents the current machine configuration.

20 The named numbers Fine_Delta and Tick are of the type *universal_real*; the others are of the type *universal_integer*.

21 The meanings of the named numbers are:

22 Min_Int — The smallest (most negative) value allowed for the expressions of a signed_integer_type_definition.

23 Max_Int — The largest (most positive) value allowed for the expressions of a signed_integer_type_definition.

24 Max_Binary_Modulus — A power of two such that it, and all lesser positive powers of two, are allowed as the modulus of a modular_type_definition.

25 Max_Nonbinary_Modulus — A value such that it, and all lesser positive integers, are allowed as the modulus of a modular_type_definition.

26 Max_Base_Digits — The largest value allowed for the requested decimal precision in a floating_point_definition.

Max_Digits The largest value allowed for the requested decimal precision in a floating_point_definition that has no real_range_specification. Max_Digits is less than or equal to Max_Base_Digits. 27

Max_Mantissa 28
The largest possible number of binary digits in the mantissa of machine numbers of a user-defined ordinary fixed point type. (The mantissa is defined in Annex G.)

Fine_Delta The smallest delta allowed in an ordinary_fixed_point_definition that has the real_range_specification **range** –1.0 .. 1.0. 29

Tick A period in seconds approximating the real time interval during which the value of Calendar.Clock remains constant. 30

Storage_Unit 31
The number of bits per storage element.

Word_Size The number of bits per word. 32

Memory_Size 33
An implementation-defined value that is intended to reflect the memory size of the configuration in storage elements.

Address is a definite, nonlimited type with preelaborable initialization (see 10.2.1). Address represents machine addresses capable of addressing individual storage elements. Null_Address is an address that is distinct from the address of any object or program unit. 34/2

Default_Bit_Order shall be a static constant. See 13.5.3 for an explanation of Bit_Order and Default_Bit_Order. 35/2

Implementation Permissions

An implementation may add additional implementation-defined declarations to package System and its children. However, it is usually better for the implementation to provide additional functionality via implementation-defined children of System. 36/2

Implementation Advice

Address should be a private type. 37

NOTE There are also some language-defined child packages of System defined elsewhere. 38

13.7.1 The Package System.Storage_Elements

Static Semantics

The following language-defined library package exists: 1

```
package System.Storage_Elements                                          2/5
   with Pure is

   type Storage_Offset is range implementation-defined;                  3

   subtype Storage_Count is Storage_Offset range 0..Storage_Offset'Last; 4

   type Storage_Element is mod implementation-defined;                   5
   for Storage_Element'Size use Storage_Unit;
   type Storage_Array is array
     (Storage_Offset range <>) of aliased Storage_Element;
   for Storage_Array'Component_Size use Storage_Unit;

   -- Address Arithmetic:                                               6
```

```
7/3         function "+"(Left : Address; Right : Storage_Offset) return Address
               with Convention => Intrinsic;
            function "+"(Left : Storage_Offset; Right : Address) return Address
               with Convention => Intrinsic;
            function "-"(Left : Address; Right : Storage_Offset) return Address
               with Convention => Intrinsic;
            function "-"(Left, Right : Address) return Storage_Offset
               with Convention => Intrinsic;
8/3         function "mod"(Left : Address; Right : Storage_Offset)
               return Storage_Offset
                  with Convention => Intrinsic;
9           -- Conversion to/from integers:
10/3        type Integer_Address is implementation-defined;
            function To_Address(Value : Integer_Address) return Address
               with Convention => Intrinsic;
            function To_Integer(Value : Address) return Integer_Address
               with Convention => Intrinsic;
11/3     end System.Storage_Elements;
```

12 Storage_Element represents a storage element. Storage_Offset represents an offset in storage elements. Storage_Count represents a number of storage elements. Storage_Array represents a contiguous sequence of storage elements.

13 Integer_Address is a (signed or modular) integer subtype. To_Address and To_Integer convert back and forth between this type and Address.

Implementation Requirements

14 Storage_Offset'Last shall be greater than or equal to Integer'Last or the largest possible storage offset, whichever is smaller. Storage_Offset'First shall be <= (–Storage_Offset'Last).

Paragraph 15 was deleted.

Implementation Advice

16 Operations in System and its children should reflect the target environment semantics as closely as is reasonable. For example, on most machines, it makes sense for address arithmetic to “wrap around”. Operations that do not make sense should raise Program_Error.

13.7.2 The Package System.Address_To_Access_Conversions

Static Semantics

1 The following language-defined generic library package exists:

```
2/5     generic
           type Object(<>) is limited private;
        package System.Address_To_Access_Conversions
           with Preelaborate, Nonblocking, Global => in out synchronized is
3/3        type Object_Pointer is access all Object;
           function To_Pointer(Value : Address) return Object_Pointer
              with Convention => Intrinsic;
           function To_Address(Value : Object_Pointer) return Address
              with Convention => Intrinsic;
4/3     end System.Address_To_Access_Conversions;
```

5/2 The To_Pointer and To_Address subprograms convert back and forth between values of types Object_Pointer and Address. To_Pointer(X'Address) is equal to X'Unchecked_Access for any X that allows Unchecked_Access. To_Pointer(Null_Address) returns **null**. For other addresses, the behavior is

unspecified. To_Address(**null**) returns Null_Address. To_Address(Y), where Y /= **null**, returns Y.**all**'Address.

Implementation Permissions

An implementation may place restrictions on instantiations of Address_To_Access_Conversions. 6

13.8 Machine Code Insertions

A machine code insertion can be achieved by a call to a subprogram whose sequence_of_statements contains code_statements. 1

Syntax

code_statement ::= qualified_expression; 2

A code_statement is only allowed in the handled_sequence_of_statements of a subprogram_body. If a subprogram_body contains any code_statements, then within this subprogram_body the only allowed form of statement is a code_statement (labeled or not), the only allowed declarative_items are use_clauses, and no exception_handler is allowed (comments and pragmas are allowed as usual). 3

Name Resolution Rules

The qualified_expression is expected to be of any type. 4

Legality Rules

The qualified_expression shall be of a type declared in package System.Machine_Code. 5

A code_statement shall appear only within the scope of a with_clause that mentions package System.Machine_Code. 6

Static Semantics

The contents of the library package System.Machine_Code (if provided) are implementation defined. The meaning of code_statements is implementation defined. Typically, each qualified_expression represents a machine instruction or assembly directive. 7

Implementation Permissions

An implementation may place restrictions on code_statements. An implementation is not required to provide package System.Machine_Code. 8

NOTE 1 An implementation can provide implementation-defined pragmas specifying register conventions and calling conventions. 9/5

NOTE 2 Machine code functions are exempt from the rule that a return statement is required. In fact, return statements are forbidden, since only code_statements are allowed. 10/2

NOTE 3 Intrinsic subprograms (see 6.3.1, “Conformance Rules”) can also be used to achieve machine code insertions. Interface to assembly language can be achieved using the features in Annex B, “Interface to Other Languages”. 11

Examples

Example of a code statement: 12

```
M : Mask;
procedure Set_Mask
  with Inline;
```
13/3

14
```
procedure Set_Mask is
  use System.Machine_Code; -- assume "with System.Machine_Code;" appears somewhere above
begin
  SI_Format'(Code => SSM, B => M'Base_Reg, D => M'Disp);
  -- Base_Reg and Disp are implementation-defined attributes
end Set_Mask;
```

13.9 Unchecked Type Conversions

1 An unchecked type conversion can be achieved by a call to an instance of the generic function Unchecked_Conversion.

Static Semantics

2 The following language-defined generic library function exists:

3/5
```
generic
   type Source(<>) is limited private;
   type Target(<>) is limited private;
function Ada.Unchecked_Conversion(S : Source) return Target
   with Pure, Nonblocking, Convention => Intrinsic;
```

Dynamic Semantics

4 The size of the formal parameter S in an instance of Unchecked_Conversion is that of its subtype. This is the actual subtype passed to Source, except when the actual is an unconstrained composite subtype, in which case the subtype is constrained by the bounds or discriminants of the value of the actual expression passed to S.

5 If all of the following are true, the effect of an unchecked conversion is to return the value of an object of the target subtype whose representation is the same as that of the source object S:

6 • S'Size = Target'Size.

7/3 • S'Alignment is a multiple of Target'Alignment or Target'Alignment is zero.

8 • The target subtype is not an unconstrained composite subtype.

9 • S and the target subtype both have a contiguous representation.

10 • The representation of S is a representation of an object of the target subtype.

11/2 Otherwise, if the result type is scalar, the result of the function is implementation defined, and can have an invalid representation (see 13.9.1). If the result type is nonscalar, the effect is implementation defined; in particular, the result can be abnormal (see 13.9.1).

Implementation Permissions

12 An implementation may return the result of an unchecked conversion by reference, if the Source type is not a by-copy type. In this case, the result of the unchecked conversion represents simply a different (read-only) view of the operand of the conversion.

13 An implementation may place restrictions on Unchecked_Conversion.

Implementation Advice

14/2 Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data.

15 The implementation should not generate unnecessary runtime checks to ensure that the representation of S is a representation of the target type. It should take advantage of the permission to return by reference

when possible. Restrictions on unchecked conversions should be avoided unless required by the target environment.

The recommended level of support for unchecked conversions is: 16

- Unchecked conversions should be supported and should be reversible in the cases where this subclause defines the result. To enable meaningful use of unchecked conversion, a contiguous representation should be used for elementary subtypes, for statically constrained array subtypes whose component subtype is one of the subtypes described in this paragraph, and for record subtypes without discriminants whose component subtypes are described in this paragraph. 17/3

13.9.1 Data Validity

Certain actions that can potentially lead to erroneous execution are not directly erroneous, but instead can cause objects to become *abnormal*. Subsequent uses of abnormal objects can be erroneous. 1

A scalar object can have an *invalid representation*, which means that the object's representation does not represent any value of the object's subtype. The primary cause of invalid representations is uninitialized variables. 2

Abnormal objects and invalid representations are explained in this subclause. 3

Dynamic Semantics

When an object is first created, and any explicit or default initializations have been performed, the object and all of its parts are in the *normal* state. Subsequent operations generally leave them normal. However, an object or part of an object can become *abnormal* in the following ways: 4

- An assignment to the object is disrupted due to an abort (see 9.8) or due to the failure of a language-defined check (see 11.6). 5
- The object is not scalar, and is passed to an **in out** or **out** parameter of an imported procedure, the Read procedure of an instance of Sequential_IO, Direct_IO, or Storage_IO, or the stream attribute T'Read, if after return from the procedure the representation of the parameter does not represent a value of the parameter's subtype. 6/2
- The object is the return object of a function call of a nonscalar type, and the function is an imported function, an instance of Unchecked_Conversion, or the stream attribute T'Input, if after return from the function the representation of the return object does not represent a value of the function's subtype. 6.1/2

For an imported object, it is the programmer's responsibility to ensure that the object remains in a normal state. 6.2/2

Whether or not an object actually becomes abnormal in these cases is not specified. An abnormal object becomes normal again upon successful completion of an assignment to the object as a whole. 7

Erroneous Execution

It is erroneous to evaluate a primary that is a name denoting an abnormal object, or to evaluate a prefix that denotes an abnormal object. 8

Bounded (Run-Time) Errors

If the representation of a scalar object does not represent a value of the object's subtype (perhaps because the object was not initialized), the object is said to have an *invalid representation*. It is a bounded error to evaluate the value of such an object. If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution continues using the invalid representation. The rules of the language outside 9

this subclause assume that all objects have valid representations. The semantics of operations on invalid representations are as follows:

10 • If the representation of the object represents a value of the object's type, the value of the type is used.

11 • If the representation of the object does not represent a value of the object's type, the semantics of operations on such representations is implementation-defined, but does not by itself lead to erroneous or unpredictable execution, or to other objects becoming abnormal.

Erroneous Execution

12/3 A call to an imported function or an instance of Unchecked_Conversion is erroneous if the result is scalar, the result object has an invalid representation, and the result is used other than as the expression of an assignment_statement or an object_declaration, as the *object*_name of an object_renaming_declaration, or as the prefix of a Valid attribute. If such a result object is used as the source of an assignment, and the assigned value is an invalid representation for the target of the assignment, then any use of the target object prior to a further assignment to the target object, other than as the prefix of a Valid attribute reference, is erroneous.

13/3 The dereference of an access value is erroneous if it does not designate an object of an appropriate type or a subprogram with an appropriate profile, if it designates a nonexistent object, or if it is an access-to-variable value that designates a constant object and it did not originate from an attribute_reference applied to an aliased variable view of a controlled or immutably limited object. An access value whose dereference is erroneous can exist, for example, because of Unchecked_Deallocation, Unchecked_Access, or Unchecked_Conversion.

14 NOTE Objects can become abnormal due to other kinds of actions that directly update the object's representation; such actions are generally considered directly erroneous, however.

13.9.2 The Valid Attribute

1 The Valid attribute can be used to check the validity of data produced by unchecked conversion, input, interface to foreign languages, and the like.

Static Semantics

2 For a prefix X that denotes a scalar object (after any implicit dereference), the following attribute is defined:

3/4 X'Valid Yields True if and only if the object denoted by X is normal, has a valid representation, and then, if the preceding conditions hold, the value of X also satisfies the predicates of the nominal subtype of X. The value of this attribute is of the predefined type Boolean.

3.1/6 Notwithstanding what this Reference Manual says elsewhere, evaluating X'Valid does not evaluate the prefix X unless X is normal and has a valid representation. If X is Volatile, evaluating X'Valid is still considered a read of X.

4 NOTE 1 Invalid data can be created in the following cases (not counting erroneous or unpredictable execution):

5 • an uninitialized scalar object,

6 • the result of an unchecked conversion,

7 • input,

8 • interface to another language (including machine code),

9 • aborting an assignment,

10 • disrupting an assignment due to the failure of a language-defined check (see 11.6), and

- use of an object whose Address has been specified. 11

NOTE 2 Since evaluating X'Valid does not evaluate X unless X is normal and valid, it is not an error to check the validity of an object that is invalid or abnormal. Determining whether X satisfies the predicates of its nominal subtype can include an evaluation of X, but this determination is made only after it has been determined that X has a valid representation and therefore is safe to evaluate. 12/6

This paragraph was deleted. 13/6

NOTE 3 The Valid attribute can be used to check the result of calling an instance of Unchecked_Conversion (or any other operation that can return invalid values). However, an exception handler is still useful because implementations are permitted to raise Constraint_Error or Program_Error if they detect the use of an invalid representation (see 13.9.1). 14/5

13.10 Unchecked Access Value Creation

The attribute Unchecked_Access is used to create access values in an unsafe manner — the programmer is responsible for preventing “dangling references”. 1

Static Semantics

The following attribute is defined for a prefix X that denotes an aliased view of an object: 2

X'Unchecked_Access 3

All rules and semantics that apply to X'Access (see 3.10.2) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package.

NOTE 1 This attribute is provided to support the situation where a local object is to be inserted into a global linked data structure, when the programmer knows that it will always be removed from the data structure prior to exiting the object's scope. The Access attribute would be illegal in this case (see 3.10.2). 4/5

NOTE 2 There is no Unchecked_Access attribute for subprograms. 5

13.11 Storage Management

Each access-to-object type has an associated storage pool. The storage allocated by an allocator comes from the pool; instances of Unchecked_Deallocation return storage to the pool. Several access types can share the same pool. 1

A storage pool is a variable of a type in the class rooted at Root_Storage_Pool, which is an abstract limited controlled type. By default, the implementation chooses a *standard storage pool* for each access-to-object type. The user may define new pool types, and may override the choice of pool for an access-to-object type by specifying Storage_Pool for the type. 2/2

Legality Rules

If Storage_Pool is specified for a given access type, Storage_Size shall not be specified for it. 3

Static Semantics

The following language-defined library package exists: 4

```
with Ada.Finalization;                                                    5/5
with System.Storage_Elements;
package System.Storage_Pools
    with Pure, Nonblocking => False is

    type Root_Storage_Pool is                                             6/5
        abstract new Ada.Finalization.Limited_Controlled with private
        with Preelaborable_Initialization;
```

7
```
procedure Allocate(
  Pool : in out Root_Storage_Pool;
  Storage_Address : out Address;
  Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
  Alignment : in Storage_Elements.Storage_Count) is abstract;
```

8
```
procedure Deallocate(
  Pool : in out Root_Storage_Pool;
  Storage_Address : in Address;
  Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
  Alignment : in Storage_Elements.Storage_Count) is abstract;
```

9
```
function Storage_Size(Pool : Root_Storage_Pool)
    return Storage_Elements.Storage_Count is abstract;
```

10
```
private
   ... -- not specified by the language
end System.Storage_Pools;
```

11 A *storage pool type* (or *pool type*) is a descendant of Root_Storage_Pool. The *elements* of a storage pool are the objects allocated in the pool by allocators.

12/2 For every access-to-object subtype S, the following representation attributes are defined:

13 S'Storage_Pool

Denotes the storage pool of the type of S. The type of this attribute is Root_Storage_-Pool'Class.

14 S'Storage_Size

Yields the result of calling Storage_Size(S'Storage_Pool), which is intended to be a measure of the number of storage elements reserved for the pool. The type of this attribute is *universal_integer*.

15/5 Storage_Size or Storage_Pool may be specified for a nonderived access-to-object type via an attribute_-definition_clause; the name in a Storage_Pool clause shall denote a variable. If the nominal subtype of the name specified for Storage_Pool is nonblocking (see 9.5), then the primitive Allocate, Deallocate, and Storage_Size subprograms of that type shall be nonblocking. Additionally, if the pool is one that supports subpools (see 13.11.4), the primitive Default_Subpool_for_Pool, Allocate_From_Subpool, and Deallocate_Subpool subprograms shall be nonblocking.

16/3 An allocator of a type *T* that does not support subpools allocates storage from *T*'s storage pool. If the storage pool is a user-defined object, then the storage is allocated by calling Allocate as described below. Allocators for types that support subpools are described in 13.11.4.

17 If Storage_Pool is not specified for a type defined by an access_to_object_definition, then the implementation chooses a standard storage pool for it in an implementation-defined manner. In this case, the exception Storage_Error is raised by an allocator if there is not enough storage. It is implementation defined whether or not the implementation provides user-accessible names for the standard pool type(s).

17.1/6 The type(s) of the standard pool(s), and the primitive Allocate, Deallocate, and Storage_Size subprograms for the standard pool(s) are nonblocking. Concurrent invocations of these subprograms do not conflict with one another (see 9.10) when applied to standard storage pools. Standard storage pool objects are considered to be synchronized objects.

18/6 If Storage_Size is specified for an access type *T*, an implementation-defined pool *P* is used for the type. The Storage_Size of *P* is at least that requested, and the storage for *P* is reclaimed when the master containing the declaration of the access type is left. If the implementation cannot satisfy the request, Storage_Error is raised at the freezing point of type *T*. The storage pool *P* is used only for allocators returning type *T* or other access types specified to use *T*'Storage_Pool. Storage_Error is raised by an allocator returning such a type if the storage space of *P* is exhausted (additional memory is not allocated).

The type of *P*, and the primitive Allocate, Deallocate, and Storage_Size subprograms of *P* are nonblocking. Concurrent invocations of these subprograms do not conflict with one another (see 9.10) when applied to *P*, and *P* is considered to be a synchronized object.

If neither Storage_Pool nor Storage_Size are specified, then the meaning of Storage_Size is implementation defined. 18.1/4

If Storage_Pool is specified for an access type, then the specified pool is used. 19

The effect of calling Allocate and Deallocate for a standard storage pool directly (rather than implicitly via an allocator or an instance of Unchecked_Deallocation) is unspecified. 20

Erroneous Execution

If Storage_Pool is specified for an access type, then if Allocate can satisfy the request, it should allocate a contiguous block of memory, and return the address of the first storage element in Storage_Address. The block should contain Size_In_Storage_Elements storage elements, and should be aligned according to Alignment. The allocated storage should not be used for any other purpose while the pool element remains in existence. If the request cannot be satisfied, then Allocate should propagate an exception (such as Storage_Error). If Allocate behaves in any other manner, then the program execution is erroneous. 21

Implementation Requirements

The Allocate procedure of a user-defined storage pool object *P* may be called by the implementation only to allocate storage for a type *T* whose pool is *P*, only at the following points: 21.1/3

- During the execution of an allocator of type *T*; 21.2/3
- During the execution of a return statement for a function whose result is built-in-place in the result of an allocator of type *T*; 21.3/3
- During the execution of an assignment operation with a target of an allocated object of type *T* with a part that has an unconstrained discriminated subtype with defaults. 21.4/3

For each of the calls of Allocate described above, *P* (equivalent to *T*'Storage_Pool) is passed as the Pool parameter. The Size_In_Storage_Elements parameter indicates the number of storage elements to be allocated, and is no more than *D*'Max_Size_In_Storage_Elements, where *D* is the designated subtype of *T*. The Alignment parameter is a nonzero integral multiple of *D*'Alignment if *D* is a specific type, and otherwise is a nonzero integral multiple of the alignment of the specific type identified by the tag of the object being created; it is unspecified if there is no such value. The Alignment parameter is no more than *D*'Max_Alignment_For_Allocation. The result returned in the Storage_Address parameter is used as the address of the allocated storage, which is a contiguous block of memory of Size_In_Storage_Elements storage elements. Any exception propagated by Allocate is propagated by the construct that contained the call. 21.5/3

The number of calls to Allocate that will be used to implement an allocator for any particular type is unspecified. The number of calls to Deallocate that will be used to implement an instance of Unchecked_Deallocation (see 13.11.2) for any particular object is the same as the number of Allocate calls for that object. 21.6/5

The Deallocate procedure of a user-defined storage pool object *P* may be called by the implementation to deallocate storage for a type *T* whose pool is *P* only at the places when an Allocate call is allowed for *P*, during the execution of an instance of Unchecked_Deallocation for *T*, or as part of the finalization of the collection of *T*. For such a call of Deallocate, *P* (equivalent to *T*'Storage_Pool) is passed as the Pool parameter. The value of the Storage_Address parameter for a call to Deallocate is the value returned in the Storage_Address parameter of the corresponding successful call to Allocate. The values of the 21.7/3

Size_In_Storage_Elements and Alignment parameters are the same values passed to the corresponding Allocate call. Any exception propagated by Deallocate is propagated by the construct that contained the call.

Documentation Requirements

22/5 An implementation shall document the set of values that a user-defined Allocate procedure has to accept for the Alignment parameter. An implementation shall document how the standard storage pool is chosen, and how storage is allocated by standard storage pools.

Implementation Advice

23 An implementation should document any cases in which it dynamically allocates heap storage for a purpose other than the evaluation of an allocator.

24 A default (implementation-provided) storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects.

25/2 The storage pool used for an allocator of an anonymous access type should be determined as follows:

25.1/2 • If the allocator is defining a coextension (see 3.10.2) of an object being created by an outer allocator, then the storage pool used for the outer allocator should also be used for the coextension;

25.2/2 • For other access discriminants and access parameters, the storage pool should be created at the point of the allocator, and be reclaimed when the allocated object becomes inaccessible;

25.3/3 • If the allocator defines the result of a function with an access result, the storage pool is determined as though the allocator were in place of the call of the function. If the call is the operand of a type conversion, the storage pool is that of the target access type of the conversion. If the call is itself defining the result of a function with an access result, this rule is applied recursively;

25.4/5 • Otherwise, a default storage pool should be created at the point where the anonymous access type is elaborated; such a storage pool may have no mechanism for the deallocation of individual objects.

26/5 NOTE 1 A user-defined storage pool type can be obtained by extending the Root_Storage_Pool type, and overriding the primitive subprograms Allocate, Deallocate, and Storage_Size. A user-defined storage pool can then be obtained by declaring an object of the type extension. The user can override Initialize and Finalize if there is any desire for nontrivial initialization and finalization for a user-defined pool type. For example, Finalize can reclaim blocks of storage that are allocated separately from the pool object itself.

27 NOTE 2 The writer of the user-defined allocation and deallocation procedures, and users of allocators for the associated access type, are responsible for dealing with any interactions with tasking. In particular:

28 • If the allocators are used in different tasks, they require mutual exclusion.

29 • If they are used inside protected objects, they cannot block.

30 • If they are used by interrupt handlers (see C.3, “Interrupt Support”), the mutual exclusion mechanism has to work properly in that context.

31 NOTE 3 The primitives Allocate, Deallocate, and Storage_Size are declared as abstract (see 3.9.3), and therefore they have to be overridden when a new (nonabstract) storage pool type is declared.

Examples

32 To associate an access type with a storage pool object, the user first declares a pool object of some type derived from Root_Storage_Pool. Then, the user defines its Storage_Pool attribute, as follows:

33
```
Pool_Object : Some_Storage_Pool_Type;
```

34
```
type T is access Designated;
for T'Storage_Pool use Pool_Object;
```

Another access type can be added to an existing storage pool, via: 35/5

```
for T2'Storage_Pool use T'Storage_Pool;
```
36

The semantics of this is implementation defined for a standard storage pool. 37

As usual, a derivative of Root_Storage_Pool can define additional operations. For example, consider the Mark_Release_Pool_Type defined in 13.11.6, that has two additional operations, Mark and Release, the following is a possible use: 38/5

```
type Mark_Release_Pool_Type
   (Pool_Size : Storage_Elements.Storage_Count)
        is new Subpools.Root_Storage_Pool_With_Subpools with private;
           -- As defined in package MR_Pool, see 13.11.6
```
39/3

```
...
```
40

```
Our_Pool : Mark_Release_Pool_Type (Pool_Size => 2000);
My_Mark : Subpool_Handle; -- As declared in 13.11.6
```
41/5

```
type Acc is access ...;
for Acc'Storage_Pool use Our_Pool;
...
```
42/3

```
My_Mark := Mark(Our_Pool);
... -- Allocate objects using "new (My_Mark) Designated(...)".
Release(My_Mark); -- Finalize objects and reclaim storage.
```
43/3

13.11.1 Storage Allocation Attributes

The Max_Size_In_Storage_Elements and Max_Alignment_For_Allocation attributes may be useful in writing user-defined pool types. 1/3

Static Semantics

For every subtype S, the following attributes are defined: 2/3

S'Max_Size_In_Storage_Elements 3/5

Denotes the maximum value for Size_In_Storage_Elements that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

S'Max_Alignment_For_Allocation 4/5

Denotes the maximum value for Alignment that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*.

For a type with access discriminants, if the implementation allocates space for a coextension in the same pool as that of the object having the access discriminant, then these attributes account for any calls on Allocate that can be performed to provide space for such coextensions. 5/5

13.11.2 Unchecked Storage Deallocation

1 Unchecked storage deallocation of an object designated by a value of an access type is achieved by a call to an instance of the generic procedure Unchecked_Deallocation.

Static Semantics

2 The following language-defined generic library procedure exists:

3/5
```
generic
   type Object(<>) is limited private;
   type Name   is access  Object;
procedure Ada.Unchecked_Deallocation(X : in out Name)
   with Preelaborate, Nonblocking,
        Global => in out Name'Storage_Pool,
        Convention => Intrinsic;
```

Legality Rules

3.1/3 A call on an instance of Unchecked_Deallocation is illegal if the actual access type of the instance is a type for which the Storage_Size has been specified by a static expression with value zero or is defined by the language to be zero. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

Dynamic Semantics

4 Given an instance of Unchecked_Deallocation declared as follows:

5
```
procedure Free is
    new Ada.Unchecked_Deallocation(
        object_subtype_name, access_to_variable_subtype_name);
```

6 Procedure Free has the following effect:

7 1. After executing Free(X), the value of X is **null**.

8 2. Free(X), when X is already equal to **null**, has no effect.

9/5 3. Free(X), when X is not equal to **null** first performs finalization of the object designated by X (and any coextensions of the object — see 3.10.2), as described in 7.6.1. It then deallocates the storage occupied by the object designated by X (and any coextensions). If the storage pool is a user-defined object, then the storage is deallocated by calling Deallocate as described in 13.11. There is one exception: if the object being freed contains tasks, it is unspecified whether the object is deallocated.

10/4 After the finalization step of Free(X), the object designated by X, and any subcomponents (and coextensions) thereof, no longer exist; their storage can be reused for other purposes.

Bounded (Run-Time) Errors

11 It is a bounded error to free a discriminated, unterminated task object. The possible consequences are:

12 • No exception is raised.

13 • Program_Error or Tasking_Error is raised at the point of the deallocation.

14 • Program_Error or Tasking_Error is raised in the task the next time it references any of the discriminants.

15 In the first two cases, the storage for the discriminants (and for any enclosing object if it is designated by an access discriminant of the task) is not reclaimed prior to task termination.

An access value that designates a nonexistent object is called a *dangling reference*. 15.1/4

If a dangling reference is dereferenced (implicitly or explicitly), execution is erroneous (see below). If there is no explicit or implicit dereference, then it is a bounded error to evaluate an expression whose result is a dangling reference. If the error is detected, either Constraint_Error or Program_Error is raised. Otherwise, execution proceeds normally, but with the possibility that the access value designates some other existing object. 15.2/4

Erroneous Execution

Evaluating a name that denotes a nonexistent object, or a protected subprogram or subprogram renaming whose associated object (if any) is nonexistent, is erroneous. The execution of a call to an instance of Unchecked_Deallocation is erroneous if the object was created other than by an allocator for an access type whose pool is Name'Storage_Pool. 16/3

Implementation Advice

For a standard storage pool, Free should actually reclaim the storage. 17

A call on an instance of Unchecked_Deallocation with a nonnull access value should raise Program_Error if the actual access type of the instance is a type for which the Storage_Size has been specified to be zero or is defined by the language to be zero. 17.1/3

NOTE 1 The rules here that refer to Free apply to any instance of Unchecked_Deallocation. 18

NOTE 2 Unchecked_Deallocation cannot be instantiated for an access-to-constant type. This is implied by the rules of 12.5.4. 19

13.11.3 Default Storage Pools

Pragma and aspect Default_Storage_Pool specify the storage pool that will be used in the absence of an explicit specification of a storage pool or storage size for an access type. 1/4

Syntax

The form of a pragma Default_Storage_Pool is as follows: 2/3

pragma Default_Storage_Pool (storage_pool_indicator); 3/3

storage_pool_indicator ::= *storage_pool_*name | **null** | Standard 3.1/4

A pragma Default_Storage_Pool is allowed immediately within the visible part of a package_specification, immediately within a declarative_part, or as a configuration pragma. 3.2/3

Name Resolution Rules

The *storage_pool_*name is expected to be of type Root_Storage_Pool'Class. 3.3/3

Legality Rules

The *storage_pool_*name shall denote a variable. 4/3

The Standard storage_pool_indicator is an identifier specific to a pragma (see 2.8) and does not denote any declaration. If the storage_pool_indicator is Standard, then there shall not be a declaration with defining_identifier Standard that is immediately visible at the point of the pragma, other than package Standard itself. 4.1/4

If the pragma is used as a configuration pragma, the storage_pool_indicator shall be either **null** or Standard, and it defines the *default pool* to be the given storage_pool_indicator within all applicable 4.2/4

compilation units (see 10.1.5), except within the immediate scope of another pragma Default_Storage_Pool. Otherwise, the pragma occurs immediately within a sequence of declarations, and it defines the default pool within the immediate scope of the pragma to be the given storage_pool_indicator, except within the immediate scope of a later pragma Default_Storage_Pool. Thus, an inner pragma overrides an outer one.

4.3/4 A pragma Default_Storage_Pool shall not be used as a configuration pragma that applies to a compilation unit that is within the immediate scope of another pragma Default_Storage_Pool.

Static Semantics

5/4 The language-defined aspect Default_Storage_Pool may be specified for a generic instance; it defines the default pool for access types within an instance. .

5.1/4 The Default_Storage_Pool aspect may be specified as Standard, which is an identifier specific to an aspect (see 13.1.1) and defines the default pool to be Standard. In this case, there shall not be a declaration with defining_identifier Standard that is immediately visible at the point of the aspect specification, other than package Standard itself.

5.2/5 Otherwise, the expected type for the Default_Storage_Pool aspect is Root_Storage_Pool'Class and the aspect_definition shall be a name that denotes a variable. This aspect overrides any Default_Storage_Pool pragma that applies to the generic unit; if the aspect is not specified, the default pool of the instance is that defined for the generic unit.

5.3/4 The effect of specifying the aspect Default_Storage_Pool on an instance of a language-defined generic unit is implementation-defined.

6/3 For nonderived access types declared in places where the default pool is defined by the pragma or aspect, their Storage_Pool or Storage_Size attribute is determined as follows, unless Storage_Pool or Storage_Size is specified for the type:

6.1/3 - If the default pool is **null**, the Storage_Size attribute is defined by the language to be zero. Therefore, an allocator for such a type is illegal.

6.2/4 - If the default pool is neither **null** nor Standard, the Storage_Pool attribute is that pool.

6.3/4 Otherwise (including when the default pool is specified as Standard), the standard storage pool is used for the type as described in 13.11.

7/3 *This paragraph was deleted.*

Implementation Permissions

8/3 An object created by an allocator that is passed as the actual parameter to an access parameter may be allocated on the stack, and automatically reclaimed, regardless of the default pool.

9/5 NOTE Default_Storage_Pool can be used with restrictions No_Coextensions and No_Access_Parameter_Allocators (see H.4) to ensure that all allocators use the default pool.

13.11.4 Storage Subpools

This subclause defines a package to support the partitioning of a storage pool into subpools. A subpool may be specified as the default to be used for allocation from the associated storage pool, or a particular subpool may be specified as part of an allocator (see 4.8). 1/3

Static Semantics

The following language-defined library package exists: 2/3

```
package System.Storage_Pools.Subpools                                                    3/5
    with Preelaborate, Global => in out synchronized is

   type Root_Storage_Pool_With_Subpools is                                               4/5
      abstract new Root_Storage_Pool with private
      with Preelaborable_Initialization;

   type Root_Subpool is abstract tagged limited private                                  5/5
      with Preelaborable_Initialization;

   type Subpool_Handle is access all Root_Subpool'Class;                                 6/3
   for Subpool_Handle'Storage_Size use 0;

   function Create_Subpool (Pool : in out Root_Storage_Pool_With_Subpools)               7/3
      return not null Subpool_Handle is abstract;

   -- The following operations are intended for pool implementers:                      8/3

   function Pool_of_Subpool (Subpool : not null Subpool_Handle)                          9/3
      return access Root_Storage_Pool_With_Subpools'Class;

   procedure Set_Pool_of_Subpool (                                                       10/5
      Subpool : in not null Subpool_Handle;
      To : in out Root_Storage_Pool_With_Subpools'Class)
         with Global => overriding in out Subpool;

   procedure Allocate_From_Subpool (                                                     11/5
      Pool : in out Root_Storage_Pool_With_Subpools;
      Storage_Address : out Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count;
      Subpool : in not null Subpool_Handle) is abstract
         with Pre'Class => Pool_of_Subpool(Subpool) = Pool'Access,
              Global => overriding in out Subpool;

   procedure Deallocate_Subpool (                                                        12/3
      Pool : in out Root_Storage_Pool_With_Subpools;
      Subpool : in out Subpool_Handle) is abstract
         with Pre'Class => Pool_of_Subpool(Subpool) = Pool'Access;

   function Default_Subpool_for_Pool (                                                   13/3
      Pool : in out Root_Storage_Pool_With_Subpools)
         return not null Subpool_Handle;

   overriding                                                                            14/3
   procedure Allocate (
      Pool : in out Root_Storage_Pool_With_Subpools;
      Storage_Address : out Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count);

   overriding                                                                            15/3
   procedure Deallocate (
      Pool : in out Root_Storage_Pool_With_Subpools;
      Storage_Address : in Address;
      Size_In_Storage_Elements : in Storage_Elements.Storage_Count;
      Alignment : in Storage_Elements.Storage_Count) is null;
```

16/3
```
      overriding
      function Storage_Size (Pool : Root_Storage_Pool_With_Subpools)
         return Storage_Elements.Storage_Count
             is (Storage_Elements.Storage_Count'Last);
```

17/3
```
   private
      ... -- not specified by the language
   end System.Storage_Pools.Subpools;
```

18/3 A *subpool* is a separately reclaimable portion of a storage pool, identified by an object of type Subpool_Handle (a *subpool handle*). A subpool handle also identifies the enclosing storage pool, a *storage pool that supports subpools*, which is a storage pool whose type is descended from Root_Storage_Pool_With_Subpools. A subpool is created by calling Create_Subpool or a similar constructor; the constructor returns the subpool handle.

19/5 A *subpool object* is an object of a type descended from Root_Subpool. Typically, subpool objects are managed by the containing storage pool; only the handles have to be exposed to clients of the storage pool. Subpool objects are designated by subpool handles, and are the run-time representation of a subpool.

20/4 Each subpool *belongs* to a single storage pool (which will always be a pool that supports subpools). An access to the pool that a subpool belongs to can be obtained by calling Pool_of_Subpool with the subpool handle. Set_Pool_of_Subpool causes the subpool of the subpool handle to belong to the given pool; this is intended to be called from subpool constructors like Create_Subpool. Set_Pool_of_Subpool propagates Program_Error if the subpool already belongs to a pool. If Set_Pool_of_Subpool has not yet been called for a subpool, Pool_of_Subpool returns **null**.

21/3 When an allocator for a type whose storage pool supports subpools is evaluated, a call is made on Allocate_From_Subpool passing in a Subpool_Handle, in addition to the parameters as defined for calls on Allocate (see 13.11). The subpool designated by the *subpool_handle*_name is used, if specified in an allocator. Otherwise, Default_Subpool_for_Pool of the Pool is used to provide a subpool handle. All requirements on the Allocate procedure also apply to Allocate_from_Subpool.

Legality Rules

22/3 If a storage pool that supports subpools is specified as the Storage_Pool for an access type, the access type is called a *subpool access type*. A subpool access type shall be a pool-specific access type.

23/3 The accessibility level of a subpool access type shall not be statically deeper than that of the storage pool object. If the specified storage pool object is a storage pool that supports subpools, then the name that denotes the object shall not denote part of a formal parameter, nor shall it denote part of a dereference of a value of a non-library-level general access type. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Dynamic Semantics

24/3 When an access type with a specified storage pool is frozen (see 13.14), if the tag of the storage pool object identifies a storage pool that supports subpools, the following checks are made:

25/3 - the name used to specify the storage pool object does not denote part of a formal parameter nor part of a dereference of a value of a non-library-level general access type; and

26/3 - the accessibility level of the access type is not deeper than that of the storage pool object.

27/3 Program_Error is raised if either of these checks fail.

28/3 A call to Subpools.Allocate(P, Addr, Size, Align) does the following:

```
Allocate_From_Subpool                                                    29/3
   (Root_Storage_Pool_With_Subpools'Class(P),
    Addr, Size, Align,
    Subpool => Default_Subpool_for_Pool
                 (Root_Storage_Pool_With_Subpools'Class(P)));
```

An allocator that allocates in a subpool raises Program_Error if the allocated object has task parts. 30/3

Unless overridden, Default_Subpool_for_Pool propagates Program_Error. 31/3

Erroneous Execution

If Allocate_From_Subpool does not meet one or more of the requirements on the Allocate procedure as given in the Erroneous Execution rules of 13.11, then the program execution is erroneous. 31.1/4

Implementation Permissions

When an allocator for a type whose storage pool is of type Root_Storage_Pool'Class is evaluated, but supports subpools, the implementation may call Allocate rather than Allocate_From_Subpool. This will have the same effect, so long as Allocate has not been overridden. 32/3

NOTE 1 A user-defined storage pool type that supports subpools can be implemented by extending the Root_Storage_Pool_With_Subpools type, and overriding the primitive subprograms Create_Subpool, Allocate_From_Subpool, and Deallocate_Subpool. Create_Subpool is expected to call Set_Pool_Of_Subpool before returning the subpool handle. To make use of such a pool, a user can declare an object of the type extension, can use it to define the Storage_Pool attribute of one or more access types, and then can invoke Create_Subpool to obtain subpool handles associated with the pool. 33/5

NOTE 2 A user-defined storage pool type that supports subpools can define additional subpool constructors similar to Create_Subpool (these typically will have additional parameters). 34/5

NOTE 3 The pool implementor can override Default_Subpool_For_Pool if they want the pool to support a default subpool for the pool. The implementor can override Deallocate if individual object reclamation is to be supported, and can override Storage_Size if there is some limit on the total size of the storage pool. The implementor can override Initialize and Finalize if there is any desire for nontrivial initialization and finalization for the pool as a whole. For example, Finalize can reclaim blocks of storage that are allocated over and above the space occupied by the pool object itself. The pool implementor can extend the Root_Subpool type as necessary to carry additional information with each subpool provided by Create_Subpool. 35/5

13.11.5 Subpool Reclamation

A subpool may be explicitly deallocated using Unchecked_Deallocate_Subpool. 1/3

Static Semantics

The following language-defined library procedure exists: 2/3

```
with System.Storage_Pools.Subpools;                                      3/6
procedure Ada.Unchecked_Deallocate_Subpool
   (Subpool : in out System.Storage_Pools.Subpools.Subpool_Handle)
   with Preelaborate, Global => in out all;
```

If Subpool is **null**, a call on Unchecked_Deallocate_Subpool has no effect. Otherwise, the subpool is finalized, and Subpool is set to **null**. 4/3

Finalization of a subpool has the following effects in the given order: 5/5

This paragraph was deleted. 6/5

1. Any of the objects allocated from the subpool that still exist are finalized in an arbitrary order; 7/3
2. All of the objects allocated from the subpool cease to exist; 7.1/4
3. The following dispatching call is then made: 8/3

9/3
```
Deallocate_Subpool(Pool_of_Subpool(Subpool).all, Subpool);
```

9.1/5 4. The subpool ceases to belong to any pool.

10/3 Finalization of a Root_Storage_Pool_With_Subpools object finalizes all subpools that belong to that pool that have not yet been finalized.

13.11.6 Storage Subpool Example

Examples

1/3 The following example is a simple but complete implementation of the classic Mark/Release pool using subpools:

2/3
```
with System.Storage_Pools.Subpools;
with System.Storage_Elements;
with Ada.Unchecked_Deallocate_Subpool;
package MR_Pool is

   use System.Storage_Pools;
      -- For uses of Subpools.
   use System.Storage_Elements;
      -- For uses of Storage_Count and Storage_Array.

   -- Mark and Release work in a stack fashion, and allocations are not allowed
   -- from a subpool other than the one at the top of the stack. This is also
   -- the default pool.

   subtype Subpool_Handle is Subpools.Subpool_Handle;

   type Mark_Release_Pool_Type (Pool_Size : Storage_Count) is new
      Subpools.Root_Storage_Pool_With_Subpools with private;

   function Mark (Pool : in out Mark_Release_Pool_Type)
      return not null Subpool_Handle;

   procedure Release (Subpool : in out Subpool_Handle) renames
      Ada.Unchecked_Deallocate_Subpool;

private

   type MR_Subpool is new Subpools.Root_Subpool with record
      Start : Storage_Count;
   end record;
   subtype Subpool_Indexes is Positive range 1 .. 10;
   type Subpool_Array is array (Subpool_Indexes) of aliased MR_Subpool;

   type Mark_Release_Pool_Type (Pool_Size : Storage_Count) is new
      Subpools.Root_Storage_Pool_With_Subpools with record
      Storage         : Storage_Array (0 .. Pool_Size);
      Next_Allocation : Storage_Count := 0;
      Markers         : Subpool_Array;
      Current_Pool    : Subpool_Indexes := 1;
   end record;

   overriding
   function Create_Subpool (Pool : in out Mark_Release_Pool_Type)
      return not null Subpool_Handle;

   function Mark (Pool : in out Mark_Release_Pool_Type)
      return not null Subpool_Handle renames Create_Subpool;

   overriding
   procedure Allocate_From_Subpool (
      Pool : in out Mark_Release_Pool_Type;
      Storage_Address : out System.Address;
      Size_In_Storage_Elements : in Storage_Count;
      Alignment : in Storage_Count;
      Subpool : not null Subpool_Handle);
```

```
   overriding                                                                    15/3
   procedure Deallocate_Subpool (
      Pool : in out Mark_Release_Pool_Type;
      Subpool : in out Subpool_Handle);

   overriding                                                                    16/3
   function Default_Subpool_for_Pool (Pool : in out Mark_Release_Pool_Type)
      return not null Subpool_Handle;

   overriding                                                                    17/3
   procedure Initialize (Pool : in out Mark_Release_Pool_Type);

   -- We don't need Finalize.                                                    18/3

end MR_Pool;                                                                     19/3

package body MR_Pool is                                                          20/3

   use type Subpool_Handle;                                                      21/3

   procedure Initialize (Pool : in out Mark_Release_Pool_Type) is                22/3
      -- Initialize the first default subpool.
   begin
      Pool.Markers(1).Start := 1;
      Subpools.Set_Pool_of_Subpool
         (Pool.Markers(1)'Unchecked_Access, Pool);
   end Initialize;

   function Create_Subpool (Pool : in out Mark_Release_Pool_Type)                23/3
      return not null Subpool_Handle is
      -- Mark the current allocation location.
   begin
      if Pool.Current_Pool = Subpool_Indexes'Last then
         raise Storage_Error; -- No more subpools.
      end if;
      Pool.Current_Pool := Pool.Current_Pool + 1; -- Move to the next subpool

      return Result : constant not null Subpool_Handle :=                        24/3
         Pool.Markers(Pool.Current_Pool)'Unchecked_Access
      do
         Pool.Markers(Pool.Current_Pool).Start := Pool.Next_Allocation;
         Subpools.Set_Pool_of_Subpool (Result, Pool);
      end return;
   end Create_Subpool;

   procedure Deallocate_Subpool (                                                25/3
      Pool : in out Mark_Release_Pool_Type;
      Subpool : in out Subpool_Handle) is
   begin
      if Subpool /= Pool.Markers(Pool.Current_Pool)'Unchecked_Access then
         raise Program_Error; -- Only the last marked subpool can be released.
      end if;
      if Pool.Current_Pool /= 1 then
         Pool.Next_Allocation := Pool.Markers(Pool.Current_Pool).Start;
         Pool.Current_Pool := Pool.Current_Pool - 1; -- Move to the previous subpool
      else -- Reinitialize the default subpool:
         Pool.Next_Allocation := 1;
         Subpools.Set_Pool_of_Subpool
            (Pool.Markers(1)'Unchecked_Access, Pool);
      end if;
   end Deallocate_Subpool;

   function Default_Subpool_for_Pool (Pool : in out Mark_Release_Pool_Type)      26/3
      return not null Subpool_Handle is
   begin
      return Pool.Markers(Pool.Current_Pool)'Unchecked_Access;
   end Default_Subpool_for_Pool;
```

27/3
```
      procedure Allocate_From_Subpool (
         Pool : in out Mark_Release_Pool_Type;
         Storage_Address : out System.Address;
         Size_In_Storage_Elements : in Storage_Count;
         Alignment : in Storage_Count;
         Subpool : not null Subpool_Handle) is
      begin
         if Subpool /= Pool.Markers(Pool.Current_Pool)'Unchecked_Access then
            raise Program_Error; -- Only the last marked subpool can be used for allocations.
         end if;
```
28/4
```
         -- Check for the maximum supported alignment, which is the alignment of the storage area:
         if Alignment > Pool.Storage'Alignment then
            raise Program_Error;
         end if;
         -- Correct the alignment if necessary:
         Pool.Next_Allocation := Pool.Next_Allocation +
            ((-Pool.Next_Allocation) mod Alignment);
         if Pool.Next_Allocation + Size_In_Storage_Elements >
            Pool.Pool_Size then
            raise Storage_Error; -- Out of space.
         end if;
         Storage_Address := Pool.Storage (Pool.Next_Allocation)'Address;
         Pool.Next_Allocation :=
            Pool.Next_Allocation + Size_In_Storage_Elements;
      end Allocate_From_Subpool;
```
29/3
```
   end MR_Pool;
```

13.12 Pragma Restrictions and Pragma Profile

1/3 A pragma Restrictions expresses the user's intent to abide by certain restrictions. A pragma Profile expresses the user's intent to abide by a set of Restrictions or other specified run-time policies. These may facilitate the construction of simpler run-time environments.

Syntax

2 The form of a pragma Restrictions is as follows:

3 **pragma** Restrictions(restriction{, restriction});

4/2 restriction ::= *restriction_*identifier
| *restriction_parameter_*identifier => restriction_parameter_argument

4.1/2 restriction_parameter_argument ::= name | expression

Name Resolution Rules

5 Unless otherwise specified for a particular restriction, the expression is expected to be of any integer type.

Legality Rules

6 Unless otherwise specified for a particular restriction, the expression shall be static, and its value shall be nonnegative.

Paragraph 7 was deleted.

Post-Compilation Rules

8/3 A pragma Restrictions is a configuration pragma. If a pragma Restrictions applies to any compilation unit included in the partition, this may impose either (or both) of two kinds of requirements, as specified for the particular restriction:

- A restriction may impose requirements on some or all of the units comprising the partition. Unless otherwise specified for a particular restriction, such a requirement applies to all of the units comprising the partition and is enforced via a post-compilation check. 8.1/3
- A restriction may impose requirements on the run-time behavior of the program, as indicated by the specification of run-time behavior associated with a violation of the requirement. 8.2/3

For the purpose of checking whether a partition contains constructs that violate any restriction (unless specified otherwise for a particular restriction): 8.3/1

- Generic instances are logically expanded at the point of instantiation; 8.4/1
- If an object of a type is declared or allocated and not explicitly initialized, then all expressions appearing in the definition for the type and any of its ancestors are presumed to be used; 8.5/1
- A default_expression for a formal parameter or a generic formal object is considered to be used if and only if the corresponding actual parameter is not provided in a given call or instantiation; 8.6/6
- An assertion policy of Check is assumed to be in effect everywhere for the assertion aspects Static_Predicate and Dynamic_Predicate, but not for other assertion aspects. Subject to this assumption, if the assertion policy in effect for the corresponding assertion aspect of an assertion expression is Ignore, then the expression is considered not to be used. 8.7/6

Implementation Permissions

An implementation may provide implementation-defined restrictions; the identifier for an implementation-defined restriction shall differ from those of the language-defined restrictions. 8.8/6

An implementation may place limitations on the values of the expression that are supported, and limitations on the supported combinations of restrictions. The consequences of violating such limitations are implementation defined. 9

An implementation is permitted to omit enforcement of restrictions for code that is recognized at compile time to be unreachable and for which no code is generated, subject to the aforementioned assumption that an Assertion_Policy of Check is in effect everywhere for the two subtype predicate assertion aspects. 9.1/6

Whenever enforcement of a restriction is not required prior to execution, an implementation may nevertheless enforce the restriction prior to execution of a partition to which the restriction applies, provided that every execution of the partition would violate the restriction. 9.2/1

Syntax

The form of a pragma Profile is as follows: 10/3

pragma Profile (*profile*_identifier {, *profile*_pragma_argument_association}); 11/3

Legality Rules

The *profile*_identifier shall be the name of a usage profile. The semantics of any *profile*_pragma_argument_associations are defined by the usage profile specified by the *profile*_identifier. 12/3

Static Semantics

A profile is equivalent to the set of configuration pragmas that is defined for each usage profile. 13/3

Post-Compilation Rules

A pragma Profile is a configuration pragma. There may be more than one pragma Profile for a partition. 14/3

Implementation Permissions

15/3 An implementation may provide implementation-defined usage profiles; the identifier for an implementation-defined usage profile shall differ from those of the language-defined usage profiles.

16/2 NOTE 1 Restrictions intended to facilitate the construction of efficient tasking run-time systems are defined in D.7. Restrictions intended for use when constructing high integrity systems are defined in H.4.

17 NOTE 2 An implementation has to enforce the restrictions in cases where enforcement is required, even if it chooses not to take advantage of the restrictions in terms of efficiency.

13.12.1 Language-Defined Restrictions and Profiles

Static Semantics

1/2 The following *restriction*_identifiers are language defined (additional restrictions are defined in the Specialized Needs Annexes):

1.1/3 No_Implementation_Aspect_Specifications
There are no implementation-defined aspects specified by an aspect_specification. This restriction applies only to the current compilation or environment, not the entire partition.

2/6 No_Implementation_Attributes
There are no implementation-defined attributes. There are no implementation-defined extensions to the definition of any language-defined attribute (see 4.1.4). This restriction applies only to the current compilation or environment, not the entire partition.

2.1/3 No_Implementation_Identifiers
There are no usage names that denote declarations with implementation-defined identifiers that occur within language-defined packages or instances of language-defined generic packages. Such identifiers can arise as follows:

2.2/3 - The following language-defined packages and generic packages allow implementation-defined identifiers:
  2.3/3 - package System (see 13.7);
  2.4/3 - package Standard (see A.1);
  2.5/3 - package Ada.Command_Line (see A.15);
  2.6/3 - package Interfaces.C (see B.3);
  2.7/3 - package Interfaces.C.Strings (see B.3.1);
  2.8/3 - package Interfaces.C.Pointers (see B.3.2);
  2.9/3 - package Interfaces.COBOL (see B.4);
  2.10/3 - package Interfaces.Fortran (see B.5);
2.11/3 - The following language-defined packages contain only implementation-defined identifiers:
  2.12/3 - package System.Machine_Code (see 13.8);
  2.13/3 - package Ada.Directories.Information (see A.16);
  2.14/3 - nested Implementation packages of the Queue containers (see A.18.28-31);
  2.15/3 - package Interfaces (see B.2);
  2.16/3 - package Ada.Interrupts.Names (see C.3.2).

For package Standard, Standard.Long_Integer and Standard.Long_Float are considered language-defined identifiers, but identifiers such as Standard.Short_Short_Integer are considered implementation-defined. 2.17/3

This restriction applies only to the current compilation or environment, not the entire partition. 2.18/3

No_Implementation_Pragmas 3/2
There are no implementation-defined pragmas or pragma arguments. This restriction applies only to the current compilation or environment, not the entire partition.

No_Implementation_Units 3.1/3
There is no mention in the context_clause of any implementation-defined descendants of packages Ada, Interfaces, or System. This restriction applies only to the current compilation or environment, not the entire partition.

No_Obsolescent_Features 4/3
There is no use of language features defined in Annex J. It is implementation defined whether uses of the renamings of J.1 and of the pragmas of J.15 are detected by this restriction. This restriction applies only to the current compilation or environment, not the entire partition.

The following *restriction_parameter_*identifiers are language defined: 5/3

No_Dependence 6/2
Specifies a library unit on which there are no semantic dependences.

No_Specification_of_Aspect 6.1/3
Identifies an aspect for which no aspect_specification, attribute_definition_clause, or pragma is given.

No_Use_Of_Attribute 6.2/3
Identifies an attribute for which no attribute_reference or attribute_definition_clause is given.

No_Use_Of_Pragma 6.3/3
Identifies a pragma which is not to be used.

No_Unrecognized_Aspects 6.4/5
There are no aspect_specifications having an unrecognized *aspect_*identifier. This restriction applies only to the current compilation or environment, not the entire partition.

No_Unrecognized_Pragmas 6.5/5
There are no pragmas having an unrecognized pragma identifier. This restriction applies only to the current compilation or environment, not the entire partition.

Legality Rules

The restriction_parameter_argument of a No_Dependence restriction shall be a name; the name shall have the form of a full expanded name of a library unit, but can be a name that has no corresponding unit currently present in the environment. 7/5

The restriction_parameter_argument of a No_Specification_of_Aspect restriction shall be an identifier; this is an identifier specific to a pragma (see 2.8) and does not denote any declaration. 7.1/3

The restriction_parameter_argument of a No_Use_Of_Attribute restriction shall be an identifier or one of the reserved words Access, Delta, Digits, Mod, or Range; this is an identifier specific to a pragma. 7.2/3

The restriction_parameter_argument of a No_Use_Of_Pragma restriction shall be an identifier or the reserved word Interface; this is an identifier specific to a pragma. 7.3/3

Post-Compilation Rules

8/3 No compilation unit included in the partition shall depend semantically on the library unit identified by the name of a No_Dependence restriction.

Static Semantics

9/3 The following *profile*_identifier is language defined:

10/3 No_Implementation_Extensions

11/3 For usage profile No_Implementation_Extensions, there shall be no *profile*_pragma_argument_associations.

12/3 The No_Implementation_Extensions usage profile is equivalent to the following restrictions:

13/3
```
No_Implementation_Aspect_Specifications,
No_Implementation_Attributes,
No_Implementation_Identifiers,
No_Implementation_Pragmas,
No_Implementation_Units.
```

13.13 Streams

1 A *stream* is a sequence of elements comprising values from possibly different types and allowing sequential access to these values. A *stream type* is a type in the class whose root type is Streams.Root_Stream_Type. A stream type may be implemented in various ways, such as an external sequential file, an internal buffer, or a network channel.

13.13.1 The Streams Subsystem

Static Semantics

1 The abstract type Root_Stream_Type is the root type of the class of stream types. The types in this class represent different kinds of streams. A new stream type is defined by extending the root type (or some other stream type), overriding the Read and Write operations, and optionally defining additional primitive subprograms, according to the requirements of the particular kind of stream. The predefined stream-oriented attributes like T'Read and T'Write make dispatching calls on the Read and Write procedures of the Root_Stream_Type. (User-defined T'Read and T'Write attributes can also make such calls, or can call the Read and Write attributes of other types.)

1.1/5 The library package Ada.Streams has the following declaration:

2/5
```
package Ada.Streams
    with Pure, Nonblocking => False is

3/2     type Root_Stream_Type is abstract tagged limited private
            with Preelaborable_Initialization;

4/1     type Stream_Element is mod implementation-defined;
        type Stream_Element_Offset is range implementation-defined;
        subtype Stream_Element_Count is
            Stream_Element_Offset range 0..Stream_Element_Offset'Last;
        type Stream_Element_Array is
            array(Stream_Element_Offset range <>) of aliased Stream_Element;

5       procedure Read(
          Stream : in out Root_Stream_Type;
          Item   : out Stream_Element_Array;
          Last   : out Stream_Element_Offset) is abstract;
```

```
      procedure Write(
        Stream : in out Root_Stream_Type;
        Item   : in Stream_Element_Array) is abstract;
```

6

```
private
   ... -- not specified by the language
end Ada.Streams;
```

7

The Read operation transfers stream elements from the specified stream to fill the array Item. Elements are transferred until Item'Length elements have been transferred, or until the end of the stream is reached. If any elements are transferred, the index of the last stream element transferred is returned in Last. Otherwise, Item'First - 1 is returned in Last. Last is less than Item'Last only if the end of the stream is reached. 8/2

The Write operation appends Item to the specified stream. 9

Three additional packages provide stream implementations that do not make use of any file operations. These packages provide the same operations, with Streams.Storage providing an abstract interface, and two child packages providing implementations of that interface. The difference is that for Streams.Storage.Bounded, the maximum storage is bounded. 10/5

The library package Ada.Streams.Storage has the following declaration: 11/5

```
package Ada.Streams.Storage                                                    12/5
   with Pure, Nonblocking is

   type Storage_Stream_Type is abstract new Root_Stream_Type with private;     13/5

   function Element_Count (Stream : Storage_Stream_Type)                       14/5
      return Stream_Element_Count is abstract;

   procedure Clear (Stream : in out Storage_Stream_Type) is abstract;          15/5

private                                                                        16/5
   ... -- not specified by the language
end Ada.Streams.Storage;
```

The library package Ada.Streams.Storage.Unbounded has the following declaration: 17/5

```
package Ada.Streams.Storage.Unbounded                                          18/5
   with Prelaborated, Nonblocking, Global => in out synchronized is

   type Stream_Type is new Storage_Stream_Type with private                     19/5
      with Default_Initial_Condition =>
          Element_Count (Stream_Type) = 0;

   overriding                                                                  20/5
   procedure Read (
      Stream : in out Stream_Type;
      Item   : out Stream_Element_Array;
      Last   : out Stream_Element_Offset)
      with Post =>
          (declare
              Num_Read : constant Stream_Element_Count :=
                 Stream_Element_Count'Min
                    (Element_Count(Stream)'Old, Item'Length);
           begin
              Last = Num_Read + Item'First - 1 and
              Element_Count (Stream) =
                 Element_Count (Stream)'Old - Num_Read);

   overriding                                                                  21/5
   procedure Write (
      Stream : in out Stream_Type;
      Item   : in Stream_Element_Array)
      with Post =>
         Element_Count (Stream) =
         Element_Count (Stream)'Old + Item'Length;
```

22/5
```
      overriding
      function Element_Count (Stream : Stream_Type)
         return Stream_Element_Count;
```

23/5
```
      overriding
      procedure Clear (Stream : in out Stream_Type)
         with Post => Element_Count (Stream) = 0;
```

24/5
```
   private
      ... -- not specified by the language
   end Ada.Streams.Storage.Unbounded;
```

25/5 The library package Ada.Streams.Storage.Bounded has the following declaration:

26/5
```
   package Ada.Streams.Storage.Bounded
       with Pure, Nonblocking is
```

27/5
```
      type Stream_Type (Max_Elements : Stream_Element_Count)
         is new Storage_Stream_Type with private
            with Default_Initial_Condition =>
               Element_Count (Stream_Type) = 0;
```

28/5
```
      overriding
      procedure Read (
         Stream : in out Stream_Type;
         Item   : out Stream_Element_Array;
         Last   : out Stream_Element_Offset)
         with Post =>
             (declare
                 Num_Read : constant Stream_Element_Count :=
                    Stream_Element_Count'Min
                       (Element_Count(Stream)'Old, Item'Length);
              begin
                 Last = Num_Read + Item'First - 1 and
                 Element_Count (Stream) =
                    Element_Count (Stream)'Old - Num_Read);
```

29/5
```
      overriding
      procedure Write (
         Stream : in out Stream_Type;
         Item   : in Stream_Element_Array)
         with Pre =>
                 Element_Count (Stream) + Item'Length <= Stream.Max_Elements
                 or else (raise Constraint_Error),
              Post =>
                 Element_Count (Stream) =
                 Element_Count (Stream)'Old + Item'Length;
```

30/5
```
      overriding
      function Element_Count (Stream : Stream_Type)
         return Stream_Element_Count
         with Post => Element_Count'Result <= Stream.Max_Elements;
```

31/5
```
      overriding
       procedure Clear (Stream : in out Stream_Type)
          with Post => Element_Count (Stream) = 0;
```

32/5
```
   private
      ... -- not specified by the language
   end Ada.Streams.Storage.Bounded;
```

33/5 The Element_Count functions return the number of stream elements that are available for reading from the given stream.

34/5 The Read and Write procedures behave as described for package Ada.Streams above. Stream elements are read in FIFO (first-in, first-out) order; stream elements are available for reading immediately after they are written.

35/5 The Clear procedures remove any available stream elements from the given stream.

Implementation Permissions

If Stream_Element'Size is not a multiple of System.Storage_Unit, then the components of Stream_Element_Array will not be aliased. 36/5

Implementation Advice

Streams.Storage.Bounded.Stream_Type objects should be implemented without implicit pointers or dynamic allocation. 37/5

NOTE 1 See A.12.1, “The Package Streams.Stream_IO” for an example of extending type Root_Stream_Type. 38

NOTE 2 If the end of stream has been reached, and Item'First is Stream_Element_Offset'First, Read will raise Constraint_Error. 39/2

13.13.2 Stream-Oriented Attributes

The type-related operational attributes Write, Read, Output, and Input convert values to a stream of elements and reconstruct values from a stream. 1/3

Static Semantics

For every subtype S of an elementary type *T*, the following representation attribute is defined: 1.1/2

S'Stream_Size 1.2/3

Denotes the number of bits read from or written to a stream by the default implementations of S'Read and S'Write. Hence, the number of stream elements required per item of elementary type *T* is:

```
T'Stream_Size / Ada.Streams.Stream_Element'Size
```
1.3/2

The value of this attribute is of type *universal_integer* and is a multiple of Stream_Element'Size. 1.4/2

Stream_Size may be specified for first subtypes via an attribute_definition_clause; the expression of such a clause shall be static, nonnegative, and a multiple of Stream_Element'Size. 1.5/2

Implementation Advice

If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size. 1.6/2

The recommended level of support for the Stream_Size attribute is: 1.7/2

- A Stream_Size clause should be supported for a discrete or fixed point type *T* if the specified Stream_Size is a multiple of Stream_Element'Size and is no less than the size of the first subtype of *T*, and no greater than the size of the largest type of the same elementary class (signed integer, modular integer, enumeration, ordinary fixed point, or decimal fixed point). 1.8/2

Static Semantics

For every subtype S of a specific type *T*, the following attributes are defined. 2

S'Write S'Write denotes a procedure with the following specification: 3

```
procedure S'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```
4/2

S'Write writes the value of *Item* to *Stream*. 5

S'Read S'Read denotes a procedure with the following specification: 6

7/2
```
procedure S'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T)
```

8 S'Read reads the value of *Item* from *Stream*.

8.1/5 *This paragraph was deleted.*

8.2/2 The default implementations of the Write and Read attributes, where available, execute as follows:

9/5 For nonderived elementary types, Read reads (and Write writes) the number of stream elements implied by the Stream_Size for the type *T*; the representation of those stream elements is implementation defined. For nonderived composite types, the Write or Read attribute for each component (excluding those, if any, that are not components of the nominal type of the object) is called in canonical order, which is last dimension varying fastest for an array (unless the convention of the array is Fortran, in which case it is first dimension varying fastest), and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included.

9.1/5 For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of the parent type or any progenitor type of *T* is available anywhere within the immediate scope of *T*, and the attribute of the parent type or the type of any of the extension components is not available at the freezing point of *T*, then the attribute of *T* shall be directly specified. For untagged derived types, the Write (resp. Read) attribute invokes the corresponding attribute of the parent type, if the attribute is available for the parent type.

9.2/5 If *T* is a discriminated type and its discriminants have defaults, then S'Read first reads the discriminants from the stream without modifying *Item*. S'Read then creates an object of type *T* constrained by these discriminants. The value of this object is then converted to the subtype of *Item* and is assigned to *Item*. Finally, the Read attribute for each nondiscriminant component of *Item* is called in canonical order as described above. Normal default initialization and finalization take place for the created object.

9.3/5 Constraint_Error is raised by the predefined Write attribute if the value of the elementary item is outside the range of values representable using Stream_Size bits. For a signed integer type, an enumeration type, or a fixed point type, the range is unsigned only if the integer code for the lower bound of the first subtype is nonnegative, and a (symmetric) signed range that covers all values of the first subtype would require more than Stream_Size bits; otherwise, the range is signed.

10 For every subtype S'Class of a class-wide type *T*'Class:

11 S'Class'Write

S'Class'Write denotes a procedure with the following specification:

12/2
```
procedure S'Class'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

13 Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item.

14 S'Class'Read S'Class'Read denotes a procedure with the following specification:

15/2
```
procedure S'Class'Read(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T'Class)
```

16 Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item.

Paragraph 17 was deleted.

Static Semantics

For every subtype S of a specific type *T*, the following attributes are defined. 18

S'Output S'Output denotes a procedure with the following specification: 19

```
procedure S'Output(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```
20/2

S'Output writes the value of *Item* to *Stream*, including any bounds or discriminants. 21

S'Input S'Input denotes a function with the following specification: 22

```
function S'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T
```
23/2

S'Input reads and returns one value from *Stream*, using any bounds or discriminants written by a corresponding S'Output to determine how much to read. 24

For an untagged derived type, the default implementation of the Output (resp. Input) attribute invokes the corresponding attribute of the parent type, if the attribute is available for the parent type. For any other type, the default implementations of the Output and Input attributes, where available, execute as follows: 25/5

- If *T* is an array type, S'Output first writes the bounds, and S'Input first reads the bounds. If *T* has discriminants without defaults, S'Output first writes the discriminants (using the Write attribute of the discriminant type for each), and S'Input first reads the discriminants (using the Read attribute of the discriminant type for each). 26/3
- S'Output then calls S'Write to write the value of *Item* to the stream. S'Input then creates an object of type *T*, with the bounds or (when without defaults) the discriminants, if any, taken from the stream, passes it to S'Read, and returns the value of the object. If *T* has discriminants, then this object is unconstrained if and only the discriminants have defaults. Normal default initialization and finalization take place for this object (see 3.3.1, 7.6, and 7.6.1). 27/3

If *T* is an abstract type, then S'Input is an abstract function. 27.1/2

For every subtype S'Class of a class-wide type *T*'Class: 28

S'Class'Output 29

S'Class'Output denotes a procedure with the following specification:

```
procedure S'Class'Output(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```
30/2

First writes the external tag of *Item* to *Stream* (by calling String'Output(*Stream*, Tags.-External_Tag(*Item*'Tag)) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S. 31/2

S'Class'Input 32

S'Class'Input denotes a function with the following specification:

```
function S'Class'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T'Class
```
33/2

First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which can raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific 34/5

type identified by the internal tag; returns that result. If the specific type identified by the internal tag is abstract, Constraint_Error is raised.

35/3 In the default implementation of Read and Input for a composite type, for each scalar component that is a discriminant or that has an implicit initial value, a check is made that the value returned by Read for the component belongs to its subtype. Constraint_Error is raised if this check fails. For other scalar components, no check is made. For each component that is of an access type, if the implementation can detect that the value returned by Read for the component is not a value of its subtype, Constraint_Error is raised. If the value is not a value of its subtype and this error is not detected, the component has an abnormal value, and erroneous execution can result (see 13.9.1). In the default implementation of Read for a composite type with defaulted discriminants, if the actual parameter of Read is constrained, a check is made that the discriminants read from the stream are equal to those of the actual parameter. Constraint_Error is raised if this check fails.

36/2 It is unspecified at which point and in which order these checks are performed. In particular, if Constraint_Error is raised due to the failure of one of these checks, it is unspecified how many stream elements have been read from the stream.

37/1 In the default implementation of Read and Input for a type, End_Error is raised if the end of the stream is reached before the reading of a value of the type is completed.

37.1/5 The Nonblocking aspect is statically True and the Global aspect is **null** for the default implementations of stream-oriented attributes for elementary types. For the default implementations of stream-oriented attributes for composite types, the value of the Nonblocking aspect is that of the first subtype, and the Global aspect defaults to that of the first subtype. A default implementation of a stream-oriented attribute that has the Nonblocking aspect statically True is considered a nonblocking region. The aspect Dispatching (see H.7.1) is Read(Stream) for the default implementations of the stream-oriented attributes Read, Read'Class, Input, and Input'Class; the aspect Dispatching is Write(Stream) for the default implementations of the stream-oriented attributes Write, Write'Class, Output, and Output'Class.

38/5 The stream-oriented attributes may be specified for any type via an attribute_definition_clause. Alternatively, each of the specific stream-oriented attributes may be specified using an aspect_specification on any type_declaration, with the aspect name being the corresponding attribute name. Each of the class-wide stream-oriented attributes may be specified using an aspect_specification for a tagged type *T* using the name of the stream-oriented attribute followed by 'Class; such class-wide aspects do not apply to other descendants of *T*. If not directly specified, a default implementation of a stream-oriented attribute is implicitly composed for a nonlimited type, and for certain limited types, as defined above.

38.1/4 The subprogram name given in such an attribute_definition_clause or aspect_specification shall statically denote a subprogram that is not an abstract subprogram. Furthermore, if a specific stream-oriented attribute is specified for an interface type, the subprogram name given in the attribute_definition_clause or aspect_specification shall statically denote a null procedure.

39/2 A stream-oriented attribute for a subtype of a specific type *T* is *available* at places where one of the following conditions is true:

40/2 - *T* is nonlimited.

41/2 - The attribute_designator is Read (resp. Write) and *T* is a limited record extension, and the attribute Read (resp. Write) is available for the parent type of *T* and for the types of all of the extension components.

42/5 - *T* is a limited untagged derived type, and the attribute is available for the parent type.

- The attribute_designator is Input (resp. Output), and *T* is a limited type, and the attribute Read (resp. Write) is available for *T*. 43/2
- The attribute has been specified via an attribute_definition_clause or aspect_specification, and the attribute_definition_clause or aspect_specification is visible. 44/5

A stream-oriented attribute for a subtype of a class-wide type *T*'Class is available at places where one of the following conditions is true: 45/2

- *T* is nonlimited; 46/2
- the attribute has been specified via an attribute_definition_clause or aspect_specification, and the attribute_definition_clause or aspect_specification is visible; or 47/5
- the corresponding attribute of *T* is available, provided that if *T* has a partial view, the corresponding attribute is available at the end of the visible part where *T* is declared. 48/2

An attribute_reference for one of the stream-oriented attributes is illegal unless the attribute is available at the place of the attribute_reference. Furthermore, an attribute_reference for *T*'Input is illegal if *T* is an abstract type. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit. 49/4

Unless available for a parent type, if any, for an untagged type having a task, protected, or explicitly limited record part, the default implementation of each of the Read, Write, Input, and Output attributes raises Program_Error and performs no other action. 49.1/5

In the parameter_and_result_profiles for the default implementations of the stream-oriented attributes, the subtype of the *Item* parameter is the base subtype of *T* if *T* is a scalar type, and the first subtype otherwise. The same rule applies to the result of the Input attribute. 50/3

For an attribute_definition_clause or aspect_specification specifying one of these attributes, the subtype of the *Item* parameter shall be the first subtype or the base subtype if scalar, and the first subtype if not scalar. The same rule applies to the result of the Input function. 51/5

A type is said to *support external streaming* if Read and Write attributes are provided for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling the representation. A limited type supports external streaming only if it has available Read and Write attributes. A type with a part that is of a nonremote access type supports external streaming only if that access type or the type of some part that includes the access type component, has Read and Write attributes that have been specified via an attribute_definition_clause, and that attribute_definition_clause is visible. An anonymous access type does not support external streaming. All other types (including remote access types, see E.2.2) support external streaming. 52/3

Erroneous Execution

If the internal tag returned by Descendant_Tag to T'Class'Input identifies a type that is not library-level and whose tag has not been created, or does not exist in the partition at the time of the call, execution is erroneous. 53/2

Implementation Requirements

For every subtype *S* of a language-defined nonlimited specific type *T*, the output generated by S'Output or S'Write shall be readable by S'Input or S'Read, respectively. This rule applies across partitions if the implementation conforms to the Distributed Systems Annex. 54/1

If Constraint_Error is raised during a call to Read because of failure of one the above checks, the implementation shall ensure that the discriminants of the actual parameter of Read are not modified. 55/3

Implementation Permissions

56/5 The number of calls performed by the predefined implementation of the stream-oriented attributes on the Read and Write operations of the stream type is unspecified. An implementation may take advantage of this permission to perform internal buffering. However, all the calls on the Read and Write operations of the stream type used to implement an explicit invocation of a stream-oriented attribute shall take place before this invocation returns. An explicit invocation is one appearing explicitly in the program text, possibly through a generic instantiation (see 12.3).

56.1/5 If *T* is a discriminated type and its discriminants have defaults, then in two cases an execution of the default implementation of S'Read is not required to create an anonymous object of type *T*: If the discriminant values that are read in are equal to the corresponding discriminant values of *Item*, then creation of a new object of type *T* may be bypassed and *Item* may be used instead. If they are not equal and *Item* is a constrained variable, then Constraint_Error may be raised at that point, before any further values are read from the stream and before the object of type *T* is created.

56.2/3 A default implementation of S'Input that calls the default implementation of S'Read may create a constrained anonymous object with discriminants that match those in the stream.

57/5 NOTE 1 For a definite subtype S of a type *T*, only *T*'Write and *T*'Read are necessary to pass an arbitrary value of the subtype through a stream. For an indefinite subtype S of a type *T*, *T*'Output and *T*'Input will normally be necessary, since *T*'Write and *T*'Read do not pass bounds, discriminants, or tags.

58 NOTE 2 User-specified attributes of S'Class are not inherited by other class-wide types descended from S.

Examples

59 *Example of user-defined Write attribute:*

60/2
```
procedure My_Write(
  Stream : not null access Ada.Streams.Root_Stream_Type'Class;
  Item   : My_Integer'Base);
for My_Integer'Write use My_Write;
```

13.14 Freezing Rules

1/3 This subclause defines a place in the program text where each declared entity becomes “frozen”. A use of an entity, such as a reference to it by name, or (for a type) an expression of the type, causes freezing of the entity in some contexts, as described below. The Legality Rules forbid certain kinds of uses of an entity in the region of text where it is frozen.

2 The *freezing* of an entity occurs at one or more places (*freezing points*) in the program text where the representation for the entity has to be fully determined. Each entity is frozen from its first freezing point to the end of the program text (given the ordering of compilation units defined in 10.1.4).

2.1/3 This subclause also defines a place in the program text where the profile of each declared callable entity becomes *frozen*. A use of a callable entity causes freezing of its profile in some contexts, as described below. At the place where the profile of a callable entity becomes frozen, the entity itself becomes frozen.

3/5 The end of a declarative_part, protected_body, or a declaration of a library package or generic library package, causes *freezing* of each entity and profile declared within it, as well as the entity itself in the case of the declaration of a library unit. A noninstance proper_body, body_stub, or entry_body causes freezing of each entity and profile declared before it within the same declarative_part.

4/1 A construct that (explicitly or implicitly) references an entity can cause the *freezing* of the entity, as defined by subsequent paragraphs. At the place where a construct causes freezing, each name, expression, implicit_dereference, or range within the construct causes freezing:

- The occurrence of a generic_instantiation causes freezing, except that a name which is a generic actual parameter whose corresponding generic formal parameter is a formal incomplete type (see 12.5.1) does not cause freezing. In addition, if a parameter of the instantiation is defaulted, the default_expression or default_name for that parameter causes freezing. 5/3
- At the occurrence of an expression_function_declaration that is a completion, the return expression of the expression function causes freezing. 5.1/4
- At the occurrence of a renames-as-body whose *callable_entity*_name denotes an expression function, the return expression of the expression function causes freezing. 5.2/4
- The occurrence of an object_declaration that has no corresponding completion causes freezing. 6
- The declaration of a record extension causes freezing of the parent subtype. 7
- The declaration of a record extension, interface type, task unit, or protected unit causes freezing of any progenitor types specified in the declaration. 7.1/2
- At the freezing point of the entity associated with an aspect_specification, any static expressions within the aspect_specification cause freezing, as do expressions or names in aspect_definitions for representation aspects, or operational aspects that have a corresponding operational attribute. Similarly, if an aspect_definition for an operational aspect, other than an assertion aspect, can affect the Name Resolution, Static Semantics, or Legality Rules of a subsequent construct, then any expressions or names within the aspect_definition cause freezing at the freezing point of the associated entity. Any static expressions within an aspect_specification also cause freezing at the end of the immediately enclosing declaration list. For the purposes of this rule, if there is no declared entity associated with an aspect_specification, the freezing point is considered to occur immediately following the aspect_specification. 7.2/5

A static expression (other than within an aspect_specification) causes freezing where it occurs. An object name or nonstatic expression causes freezing where it occurs, unless the name or expression is part of a default_expression, a default_name, the return expression of an expression function, an aspect_specification, or a per-object expression of a component's constraint, in which case, the freezing occurs later as part of another construct or at the freezing point of an associated entity. 8/4

An implicit call freezes the same entities and profiles that would be frozen by an explicit call. This is true even if the implicit call is removed via implementation permissions. 8.1/3

If an expression is implicitly converted to a type or subtype *T*, then at the place where the expression causes freezing, *T* is frozen. 8.2/1

The following rules define which entities are frozen at the place where a construct causes freezing: 9

- At the place where an expression causes freezing, the type of the expression is frozen, unless the expression is an enumeration literal used as a discrete_choice of the array_aggregate of an enumeration_representation_clause or as the aspect_definition of a specification for aspect Default_Value. 10/5
- At the place where a function call causes freezing, the profile of the function is frozen. Furthermore, if a parameter of the call is defaulted, the default_expression for that parameter causes freezing. If the function call is to an expression function, the return expression of the expression function causes freezing. 10.1/4
- At the place where a generic_instantiation causes freezing of a callable entity, the profile of that entity is frozen unless the formal subprogram corresponding to the callable entity has a parameter or result of a formal untagged incomplete type; if the callable entity is an expression function, the return expression of the expression function causes freezing. 10.2/4

10.3/4 • At the place where a use of the Access or Unchecked_Access attribute whose prefix denotes an expression function causes freezing, the return expression of the expression function causes freezing.

11 • At the place where a name causes freezing, the entity denoted by the name is frozen, unless the name is a prefix of an expanded name; at the place where an object name causes freezing, the nominal subtype associated with the name is frozen.

11.1/1 • At the place where an implicit_dereference causes freezing, the nominal subtype associated with the implicit_dereference is frozen.

12 • At the place where a range causes freezing, the type of the range is frozen.

13 • At the place where an allocator causes freezing, the designated subtype of its type is frozen. If the type of the allocator is a derived type, then all ancestor types are also frozen.

14/3 • At the place where a profile is frozen, each subtype of the profile is frozen. If the corresponding callable entity is a member of an entry family, the index subtype of the family is frozen.

15/5 • At the place where a subtype is frozen, its type is frozen. At the place where a type is frozen, any expressions or names within the full type definition cause freezing, other than those that occur within an access_type_definition or an access_definition; the first subtype, and any component subtypes, index subtypes, and parent subtype of the type are frozen as well. For a specific tagged type, the corresponding class-wide type is frozen as well. For a class-wide type, the corresponding specific type is frozen as well.

15.1/6 • At the place where a specific tagged type is frozen, the primitive subprograms of the type are frozen; for each such primitive subprogram that is an expression function, its return expression causes freezing. At the place where a type is frozen, any subprogram named in an attribute_definition_clause for the type is frozen.

15.2/5 •

15.3/5 Notwithstanding the rest of this subclause, freezing an incomplete view has no effect.

Legality Rules

16 The explicit declaration of a primitive subprogram of a tagged type shall occur before the type is frozen (see 3.9.2).

17 A type shall be completely defined before it is frozen (see 3.11.1 and 7.3).

18 The completion of a deferred constant declaration shall occur before the constant is frozen (see 7.4).

19/1 An operational or representation item that directly specifies an aspect of an entity shall appear before the entity is frozen (see 13.1).

Dynamic Semantics

20/2 The tag (see 3.9) of a tagged type T is created at the point where T is frozen.

The Standard Libraries

Annex A
(normative)
Predefined Language Environment

This Annex contains the specifications of library units that shall be provided by every implementation. There are three root library units: Ada, Interfaces, and System; other library units are children of these: 1

2/5

- Standard — A.1
  - Ada — A.2
    - Assertions — 11.4.2
    - Asynchronous_Task_Control — D.11
    - Calendar — 9.6
      - Arithmetic — 9.6.1
      - Formatting — 9.6.1
      - Time_Zones — 9.6.1
    - Characters — A.3.1
      - Conversions — A.3.4
      - Handling — A.3.2
      - Latin_1 — A.3.3
    - Command_Line — A.15
    - Complex_Text_IO — G.1.3
    - Containers — A.18.1
      - Bounded_Doubly_Linked_Lists — A.18.20
      - Bounded_Hashed_Maps — A.18.21
      - Bounded_Hashed_Sets — A.18.23
      - Bounded_Indefinite_Holders — A.18.32
      - Bounded_Multiway_Trees — A.18.25
      - Bounded_Ordered_Maps — A.18.22
      - Bounded_Ordered_Sets — A.18.24
      - Bounded_Priority_Queues — A.18.31
      - Bounded_Synchronized_Queues — A.18.29
      - Bounded_Vectors — A.18.19
      - Doubly_Linked_Lists — A.18.3
      - Generic_Array_Sort — A.18.26
      - Generic_Constrained_Array_Sort — A.18.26
      - Generic_Sort — A.18.26
      - Hashed_Maps — A.18.5
      - Hashed_Sets — A.18.8
      - Indefinite_Doubly_Linked_Lists — A.18.12
      - Indefinite_Hashed_Maps — A.18.13
      - Indefinite_Hashed_Sets — A.18.15

- Standard (*...continued*)
  - Ada (*...continued*)
    - Containers (*...continued*)
      - Indefinite_Holders — A.18.18
      - Indefinite_Multiway_Trees — A.18.17
      - Indefinite_Ordered_Maps — A.18.14
      - Indefinite_Ordered_Sets — A.18.16
      - Indefinite_Vectors — A.18.11
      - Multiway_Trees — A.18.10
      - Ordered_Maps — A.18.6
      - Ordered_Sets — A.18.9
      - Synchronized_Queue_Interfaces — A.18.27
      - Unbounded_Priority_Queues — A.18.30
      - Unbounded_Synchronized_Queues — A.18.28
      - Vectors — A.18.2
    - Decimal — F.2
    - Direct_IO — A.8.4
    - Directories — A.16
      - Hierarchical_File_Names — A.16.1
      - Information — A.16
    - Dispatching — D.2.1
      - EDF — D.2.6
      - Non_Preemptive — D.2.4
      - Round_Robin — D.2.5
    - Dynamic_Priorities — D.5.1
    - Environment_Variables — A.17
    - Exceptions — 11.4.1
    - Execution_Time — D.14
      - Group_Budgets — D.14.2
      - Interrupts — D.14.3
      - Timers — D.14.1
    - Finalization — 7.6
    - Float_Text_IO — A.10.9
    - Float_Wide_Text_IO — A.11
    - Float_Wide_Wide_Text_IO — A.11

Standard (*...continued*)
Ada (*...continued*)
Integer_Text_IO — A.10.8
Integer_Wide_Text_IO — A.11
Integer_Wide_Wide_Text_IO — A.11
Interrupts — C.3.2
Names — C.3.2
IO_Exceptions — A.13
Iterator_Interfaces — 5.5.1
Locales — A.19 Numerics — A.5
Big_Numbers — A.5.5
Big_Integers — A.5.6
Big_Reals — A.5.7
Complex_Arrays — G.3.2
Complex_Elementary_Functions — G.1.2
Complex_Types — G.1.1
Discrete_Random — A.5.2
Elementary_Functions — A.5.1
Float_Random — A.5.2
Generic_Complex_Arrays — G.3.2
Generic_Complex_Elementary_Functions — G.1.2
Generic_Complex_Types — G.1.1
Generic_Elementary_Functions — A.5.1
Generic_Real_Arrays — G.3.1
Real_Arrays — G.3.1
Real_Time — D.8
Timing_Events — D.15
Sequential_IO — A.8.1
Storage_IO — A.9
Streams — 13.13.1
Storage_Streams — 13.13.1
Bounded_FIFO_Streams — 13.13.1
FIFO_Streams — 13.13.1
Stream_IO — A.12.1
Strings — A.4.1
Bounded — A.4.4
Equal_Case_Insensitive — A.4.10
Hash — A.4.9
Hash_Case_Insensitive — A.4.9
Less_Case_Insensitive — A.4.10
Equal_Case_Insensitive — A.4.10
Fixed — A.4.3
Equal_Case_Insensitive — A.4.10
Hash — A.4.9
Hash_Case_Insensitive — A.4.9
Less_Case_Insensitive — A.4.10

Standard (*...continued*)
Ada (*...continued*)
Strings (*...continued*)
Hash — A.4.9
Hash_Case_Insensitive — A.4.9
Less_Case_Insensitive — A.4.10
Maps — A.4.2
Constants — A.4.6
Text_Buffers — A.4.12
Bounded — A.4.12
Unbounded — A.4.12
Unbounded — A.4.5
Equal_Case_Insensitive — A.4.10
Hash — A.4.9
Hash_Case_Insensitive — A.4.9
Less_Case_Insensitive — A.4.10
UTF_Encoding — A.4.11
Conversions — A.4.11
Strings — A.4.11
Wide_Strings — A.4.11
Wide_Wide_Strings — A.4.11
Wide_Bounded — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Fixed — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Maps — A.4.7
Wide_Constants — A.4.7
Wide_Unbounded — A.4.7
Wide_Equal_Case_Insensitive — A.4.7
Wide_Hash — A.4.7
Wide_Hash_Case_Insensitive — A.4.7
Wide_Wide_Bounded — A.4.8
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8

Standard (*...continued*)
Ada (*...continued*)

Strings (*...continued*)
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Fixed — A.4.8
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8
Wide_Wide_Maps — A.4.8
Wide_Wide_Constants — A.4.8
Wide_Wide_Unbounded — A.4.8
Wide_Wide_Equal_Case_Insensitive — A.4.8
Wide_Wide_Hash — A.4.8
Wide_Wide_Hash_Case_Insensitive — A.4.8
Synchronous_Barriers — D.10.1
Synchronous_Task_Control — D.10
EDF — D.10
Tags — 3.9
Generic_Dispatching_Constructor — 3.9
Task_Attributes — C.7.2
Task_Identification — C.7.1
Task_Termination — C.7.3
Text_IO — A.10.1
Bounded_IO — A.10.11
Complex_IO — G.1.3
Editing — F.3.3
Text_Streams — A.12.2
Unbounded_IO — A.10.12
Unchecked_Conversion — 13.9
Unchecked_Deallocate_Subpool — 13.11.5
Unchecked_Deallocation — 13.11.2
Wide_Characters — A.3.1
Handling — A.3.5
Wide_Command_Line — A.15.1
Wide_Directories — A.16.2
Wide_Environment_Variables — A.17.1

Standard (*...continued*)
Ada (*...continued*)
Wide_Text_IO — A.11
Complex_IO — G.1.4
Editing — F.3.4
Text_Streams — A.12.3
Wide_Bounded_IO — A.11
Wide_Unbounded_IO — A.11
Wide_Wide_Characters — A.3.1
Handling — A.3.6
Wide_Wide_Command_Line — A.15.1
Wide_Wide_Directories — A.16.2
Wide_Wide_Environment_Variables — A.17.1
Wide_Wide_Text_IO — A.11
Complex_IO — G.1.5
Editing — F.3.5
Text_Streams — A.12.4
Wide_Wide_Bounded_IO — A.11
Wide_Wide_Unbounded_IO — A.11

Interfaces — B.2
C — B.3
Pointers — B.3.2
Strings — B.3.1
COBOL — B.4
Fortran — B.5

System — 13.7
Address_To_Access_Conversions — 13.7.2
Atomic_Operations — C.6.1
Exchange — C.6.2
Integer_Arithmetic — C.6.4
Modular_Arithmetic — C.6.5
Test_And_Set — C.6.3
Machine_Code — 13.8
Multiprocessors — D.16
Dispatching_Domains — D.16.1
RPC — E.5
Storage_Elements — 13.7.1
Storage_Pools — 13.11
Subpools — 13.11.4

Implementation Requirements

The implementation shall ensure that concurrent calls on any two (possibly the same) language-defined subprograms perform as specified, so long as all pairs of objects (one from each call) that are either denoted by parameters that can be passed by reference, or are designated by parameters of an access type, are nonoverlapping. 3/5

3.1/4 For the purpose of determining whether concurrent calls on text input-output subprograms are required to perform as specified above, when calling a subprogram within Text_IO or its children that implicitly operates on one of the default input-output files, the subprogram is considered to have a parameter of Current_Input or Current_Output (as appropriate).

3.2/3 If a descendant of a language-defined tagged type is declared, the implementation shall ensure that each inherited language-defined subprogram behaves as described in this Reference Manual. In particular, overriding a language-defined subprogram shall not alter the effect of any inherited language-defined subprogram.

Implementation Permissions

4 The implementation may restrict the replacement of language-defined compilation units. The implementation may restrict children of language-defined library units (other than Standard).

A.1 The Package Standard

1/3 This subclause outlines the specification of the package Standard containing all predefined identifiers in the language. The corresponding package body is not specified by the language.

2 The operators that are predefined for the types declared in the package Standard are given in comments since they are implicitly declared. Italics are used for pseudo-names of anonymous types (such as *root_real*) and for undefined information (such as *implementation-defined*).

Static Semantics

3 The library package Standard has the following declaration:

4/5
```
package Standard
   with Pure is
```

5
```
   type Boolean is (False, True);
```

6
```
   -- The predefined relational operators for this type are as follows:
```

7/1
```
   -- function "="   (Left, Right : Boolean'Base) return Boolean;
   -- function "/="  (Left, Right : Boolean'Base) return Boolean;
   -- function "<"   (Left, Right : Boolean'Base) return Boolean;
   -- function "<="  (Left, Right : Boolean'Base) return Boolean;
   -- function ">"   (Left, Right : Boolean'Base) return Boolean;
   -- function ">="  (Left, Right : Boolean'Base) return Boolean;
```

8
```
   -- The predefined logical operators and the predefined logical
   -- negation operator are as follows:
```

9/1
```
   -- function "and" (Left, Right : Boolean'Base) return Boolean'Base;
   -- function "or"  (Left, Right : Boolean'Base) return Boolean'Base;
   -- function "xor" (Left, Right : Boolean'Base) return Boolean'Base;
```

10/1
```
   -- function "not" (Right : Boolean'Base) return Boolean'Base;
```

11/2
```
   -- The integer type root_integer and the
   -- corresponding universal type universal_integer are predefined.
```

12
```
   type Integer is range implementation-defined;
```

13
```
   subtype Natural  is Integer range 0 .. Integer'Last;
   subtype Positive is Integer range 1 .. Integer'Last;
```

```
-- The predefined operators for type Integer are as follows:                    14

-- function "="  (Left, Right : Integer'Base) return Boolean;                   15
-- function "/=" (Left, Right : Integer'Base) return Boolean;
-- function "<"  (Left, Right : Integer'Base) return Boolean;
-- function "<=" (Left, Right : Integer'Base) return Boolean;
-- function ">"  (Left, Right : Integer'Base) return Boolean;
-- function ">=" (Left, Right : Integer'Base) return Boolean;

-- function "+"   (Right : Integer'Base) return Integer'Base;                   16
-- function "-"   (Right : Integer'Base) return Integer'Base;
-- function "abs" (Right : Integer'Base) return Integer'Base;

-- function "+"   (Left, Right : Integer'Base) return Integer'Base;             17
-- function "-"   (Left, Right : Integer'Base) return Integer'Base;
-- function "*"   (Left, Right : Integer'Base) return Integer'Base;
-- function "/"   (Left, Right : Integer'Base) return Integer'Base;
-- function "rem" (Left, Right : Integer'Base) return Integer'Base;
-- function "mod" (Left, Right : Integer'Base) return Integer'Base;

-- function "**"  (Left : Integer'Base; Right : Natural)                        18
--                  return Integer'Base;

-- The specification of each operator for the type                              19
-- root_integer, or for any additional predefined integer
-- type, is obtained by replacing Integer by the name of the type
-- in the specification of the corresponding operator of the type
-- Integer. The right operand of the exponentiation operator
-- remains as subtype Natural.

-- The floating point type root_real and the                                    20/2
-- corresponding universal type universal_real are predefined.

type Float is digits implementation-defined;                                    21

-- The predefined operators for this type are as follows:                       22

-- function "="   (Left, Right : Float) return Boolean;                        23
-- function "/="  (Left, Right : Float) return Boolean;
-- function "<"   (Left, Right : Float) return Boolean;
-- function "<="  (Left, Right : Float) return Boolean;
-- function ">"   (Left, Right : Float) return Boolean;
-- function ">="  (Left, Right : Float) return Boolean;

-- function "+"   (Right : Float) return Float;                                 24
-- function "-"   (Right : Float) return Float;
-- function "abs" (Right : Float) return Float;

-- function "+"   (Left, Right : Float) return Float;                           25
-- function "-"   (Left, Right : Float) return Float;
-- function "*"   (Left, Right : Float) return Float;
-- function "/"   (Left, Right : Float) return Float;

-- function "**"  (Left : Float; Right : Integer'Base) return Float;            26

-- The specification of each operator for the type root_real, or for            27
-- any additional predefined floating point type, is obtained by
-- replacing Float by the name of the type in the specification of the
-- corresponding operator of the type Float.

-- In addition, the following operators are predefined for the root             28
-- numeric types:

function "*" (Left : root_integer; Right : root_real)                           29
  return root_real;

function "*" (Left : root_real;    Right : root_integer)                        30
  return root_real;

function "/" (Left : root_real;    Right : root_integer)                        31
  return root_real;

-- The type universal_fixed is predefined.                                      32
-- The only multiplying operators defined between
-- fixed point types are
```

```
33          function "*" (Left : universal_fixed; Right : universal_fixed)
              return universal_fixed;
34          function "/" (Left : universal_fixed; Right : universal_fixed)
              return universal_fixed;
34.1/2      -- The type universal_access is predefined.
            -- The following equality operators are predefined:
34.2/2      function "="  (Left, Right: universal_access) return Boolean;
            function "/=" (Left, Right: universal_access) return Boolean;
35/3            -- The declaration of type Character is based on the standard ISO 8859-1 character set.

                -- There are no character literals corresponding to the positions for control characters.
                -- They are indicated in italics in this definition. See 3.5.2.

            type Character is
              (nul,    soh,    stx,    etx,    eot,    enq,    ack,    bel,    --0 (16#00#) .. 7 (16#07#)
               bs,     ht,     lf,     vt,     ff,     cr,     so,     si,     --8 (16#08#) .. 15 (16#0F#)

               dle,    dc1,    dc2,    dc3,    dc4,    nak,    syn,    etb,    --16 (16#10#) .. 23 (16#17#)
               can,    em,     sub,    esc,    fs,     gs,     rs,     us,     --24 (16#18#) .. 31 (16#1F#)

               ' ',    '!',    '"',    '#',    '$',    '%',    '&',    ''',    --32 (16#20#) .. 39 (16#27#)
               '(',    ')',    '*',    '+',    ',',    '-',    '.',    '/',    --40 (16#28#) .. 47 (16#2F#)

               '0',    '1',    '2',    '3',    '4',    '5',    '6',    '7',    --48 (16#30#) .. 55 (16#37#)
               '8',    '9',    ':',    ';',    '<',    '=',    '>',    '?',    --56 (16#38#) .. 63 (16#3F#)

               '@',    'A',    'B',    'C',    'D',    'E',    'F',    'G',    --64 (16#40#) .. 71 (16#47#)
               'H',    'I',    'J',    'K',    'L',    'M',    'N',    'O',    --72 (16#48#) .. 79 (16#4F#)

               'P',    'Q',    'R',    'S',    'T',    'U',    'V',    'W',    --80 (16#50#) .. 87 (16#57#)
               'X',    'Y',    'Z',    '[',    '\',    ']',    '^',    '_',    --88 (16#58#) .. 95 (16#5F#)

               '`',    'a',    'b',    'c',    'd',    'e',    'f',    'g',    --96 (16#60#) .. 103 (16#67#)
               'h',    'i',    'j',    'k',    'l',    'm',    'n',    'o',    --104 (16#68#) .. 111 (16#6F#)

               'p',    'q',    'r',    's',    't',    'u',    'v',    'w',    --112 (16#70#) .. 119 (16#77#)
               'x',    'y',    'z',    '{',    '|',    '}',    '~',    del,    --120 (16#78#) .. 127 (16#7F#)

               reserved_128,   reserved_129,   bph,    nbh,                    --128 (16#80#) .. 131 (16#83#)
               reserved_132,   nel,    ssa,    esa,                            --132 (16#84#) .. 135 (16#87#)
               hts,    htj,    vts,    pld,    plu,    ri,     ss2,    ss3,    --136 (16#88#) .. 143 (16#8F#)

               dcs,    pu1,    pu2,    sts,    cch,    mw,     spa,    epa,    --144 (16#90#) .. 151 (16#97#)
               sos,    reserved_153,   sci,    csi,                            --152 (16#98#) .. 155 (16#9B#)
               st,     osc,    pm,     apc,                                    --156 (16#9C#) .. 159 (16#9F#)

               ' ',    '¡',    '¢',    '£',    '¤',    '¥',    '¦',    '§',    --160 (16#A0#) .. 167 (16#A7#)
               '¨',    '©',    'ª',    '«',                                    --168 (16#A8#) .. 171 (16#AB#)
               '¬',    soft_hyphen,    '®',    '¯',                            --172 (16#AC#) .. 175 (16#AF#)

               '°',    '±',    '²',    '³',    '´',    'µ',    '¶',    '·',    --176 (16#B0#) .. 183 (16#B7#)
               '¸',    '¹',    'º',    '»',    '¼',    '½',    '¾',    '¿',    --184 (16#B8#) .. 191 (16#BF#)

               'À',    'Á',    'Â',    'Ã',    'Ä',    'Å',    'Æ',    'Ç',    --192 (16#C0#) .. 199 (16#C7#)
               'È',    'É',    'Ê',    'Ë',    'Ì',    'Í',    'Î',    'Ï',    --200 (16#C8#) .. 207 (16#CF#)

               'Ð',    'Ñ',    'Ò',    'Ó',    'Ô',    'Õ',    'Ö',    '×',    --208 (16#D0#) .. 215 (16#D7#)
               'Ø',    'Ù',    'Ú',    'Û',    'Ü',    'Ý',    'Þ',    'ß',    --216 (16#D8#) .. 223 (16#DF#)

               'à',    'á',    'â',    'ã',    'ä',    'å',    'æ',    'ç',    --224 (16#E0#) .. 231 (16#E7#)
               'è',    'é',    'ê',    'ë',    'ì',    'í',    'î',    'ï',    --232 (16#E8#) .. 239 (16#EF#)

               'ð',    'ñ',    'ò',    'ó',    'ô',    'õ',    'ö',    '÷',    --240 (16#F0#) .. 247 (16#F7#)
               'ø',    'ù',    'ú',    'û',    'ü',    'ý',    'þ',    'ÿ');   --248 (16#F8#) .. 255 (16#FF#)
```

```
   -- The predefined operators for the type Character are the same as for                    36
   -- any enumeration type.

   -- The declaration of type Wide_Character is based on the standard Unicode, version 15.0.0 BMP character  36.1/6
   -- set. The first 256 positions have the same contents as type Character. See 3.5.2.

   type Wide_Character is (nul, soh ... Hex_0000FFFE, Hex_0000FFFF);

   -- The declaration of type Wide_Wide_Character is based on the full                        36.2/6
   -- Unicode Standard, version 15.0.0 character set. The first 65536 positions have the
   -- same contents as type Wide_Character. See 3.5.2.

   type Wide_Wide_Character is (nul, soh ... Hex_7FFFFFFE, Hex_7FFFFFFF);
   for Wide_Wide_Character'Size use 32;

   package ASCII is ... end ASCII;  --Obsolescent; see J.5                                  36.3/2


   -- Predefined string types:                                                             37/3

   type String is array(Positive range <>) of Character
      with Pack;

   -- The predefined operators for this type are as follows:                               38

   --      function "="  (Left, Right: String) return Boolean;                           39
   --      function "/=" (Left, Right: String) return Boolean;
   --      function "<"  (Left, Right: String) return Boolean;
   --      function "<=" (Left, Right: String) return Boolean;
   --      function ">"  (Left, Right: String) return Boolean;
   --      function ">=" (Left, Right: String) return Boolean;

   --      function "&" (Left: String;    Right: String)    return String;               40
   --      function "&" (Left: Character; Right: String)    return String;
   --      function "&" (Left: String;    Right: Character) return String;
   --      function "&" (Left: Character; Right: Character) return String;

   type Wide_String is array(Positive range <>) of Wide_Character                          41/3
      with Pack;

   -- The predefined operators for this type correspond to those for String.               42

   type Wide_Wide_String is array (Positive range <>)                                      42.1/3
      of Wide_Wide_Character
         with Pack;

   -- The predefined operators for this type correspond to those for String.               42.2/2

   type Duration is delta implementation-defined range implementation-defined;             43

      -- The predefined operators for the type Duration are the same as for                44
      -- any fixed point type.

   -- The predefined exceptions:                                                           45

   Constraint_Error: exception;                                                            46
   Program_Error   : exception;
   Storage_Error   : exception;
   Tasking_Error   : exception;

end Standard;                                                                              47
```

Standard has no private part. 48

In each of the types Character, Wide_Character, and Wide_Wide_Character, the character literals for the space character (position 32) and the non-breaking space character (position 160) correspond to different values. Unless indicated otherwise, each occurrence of the character literal ' ' in this Reference Manual refers to the space character. Similarly, the character literals for hyphen (position 45) and soft hyphen (position 173) correspond to different values. Unless indicated otherwise, each occurrence of the character literal '–' in this Reference Manual refers to the hyphen character. 49/2

Dynamic Semantics

50 Elaboration of the body of Standard has no effect.

Implementation Permissions

51 An implementation may provide additional predefined integer types and additional predefined floating point types. Some or all of these types may be anonymous.

Implementation Advice

52 If an implementation provides additional named predefined integer types, then the names should end with "Integer" as in "Long_Integer". If an implementation provides additional named predefined floating point types, then the names should end with "Float" as in "Long_Float".

53 NOTE 1 Certain aspects of the predefined entities cannot be completely described in the language itself. For example, although the enumeration type Boolean can be written showing the two enumeration literals False and True, the short-circuit control forms cannot be expressed in the language.

54 NOTE 2 As explained in 8.1, "Declarative Region" and 10.1.4, "The Compilation Process", the declarative region of the package Standard encloses every library unit and consequently the main subprogram; the declaration of every library unit is assumed to occur within this declarative region. Library_items are assumed to be ordered in such a way that there are no forward semantic dependences. However, as explained in 8.3, "Visibility", the only library units that are visible within a given compilation unit are the library units named by all with_clauses that apply to the given unit, and moreover, within the declarative region of a given library unit, that library unit itself.

55 NOTE 3 If all block_statements of a program are named, then the name of each program unit can always be written as an expanded name starting with Standard (unless Standard is itself hidden). The name of a library unit cannot be a homograph of a name (such as Integer) that is already declared in Standard.

56 NOTE 4 The exception Standard.Numeric_Error is defined in J.6.

A.2 The Package Ada

Static Semantics

1 The following language-defined library package exists:

2/5
```
package Ada
   with Pure is
end Ada;
```

3/5 Ada serves as the parent of most of the other language-defined library units; its declaration is empty.

Legality Rules

4 In the standard mode, it is illegal to compile a child of package Ada.

A.3 Character Handling

1/3 This subclause presents the packages related to character processing: an empty declared pure package Characters and child packages Characters.Handling and Characters.Latin_1. The package Characters.Handling provides classification and conversion functions for Character data, and some simple functions for dealing with Wide_Character and Wide_Wide_Character data. The child package Characters.Latin_1 declares a set of constants initialized to values of type Character.

A.3.1 The Packages Characters, Wide_Characters, and Wide_Wide_Characters

Static Semantics

The library package Characters has the following declaration: 1

```
package Ada.Characters                                                    2/5
   with Pure is
end Ada.Characters;
```

The library package Wide_Characters has the following declaration: 3/2

```
package Ada.Wide_Characters                                               4/5
   with Pure is
end Ada.Wide_Characters;
```

The library package Wide_Wide_Characters has the following declaration: 5/2

```
package Ada.Wide_Wide_Characters                                          6/5
   with Pure is
end Ada.Wide_Wide_Characters;
```

Implementation Advice

If an implementation chooses to provide implementation-defined operations on Wide_Character or Wide_String (such as collating and sorting, etc.) it should do so by providing child units of Wide_Characters. Similarly if it chooses to provide implementation-defined operations on Wide_Wide_Character or Wide_Wide_String it should do so by providing child units of Wide_Wide_Characters. 7/3

A.3.2 The Package Characters.Handling

Static Semantics

The library package Characters.Handling has the following declaration: 1

```
with Ada.Characters.Conversions;                                          2/5
package Ada.Characters.Handling
  with Pure is

  --Character classification functions                                    3

  function Is_Control           (Item : in Character) return Boolean;     4/5
  function Is_Graphic           (Item : in Character) return Boolean;
  function Is_Letter            (Item : in Character) return Boolean;
  function Is_Lower             (Item : in Character) return Boolean;
  function Is_Upper             (Item : in Character) return Boolean;
  function Is_Basic             (Item : in Character) return Boolean;
  function Is_Digit             (Item : in Character) return Boolean;
  function Is_Decimal_Digit     (Item : in Character) return Boolean
                   renames Is_Digit;
  function Is_Hexadecimal_Digit (Item : in Character) return Boolean;
  function Is_Alphanumeric      (Item : in Character) return Boolean;
  function Is_Special           (Item : in Character) return Boolean;
  function Is_Line_Terminator   (Item : in Character) return Boolean;
  function Is_Mark              (Item : in Character) return Boolean;
  function Is_Other_Format      (Item : in Character) return Boolean;
  function Is_Punctuation_Connector (Item : in Character) return Boolean;
  function Is_Space             (Item : in Character) return Boolean;
  function Is_NFKC              (Item : in Character) return Boolean;
```

```
   --Conversion functions for Character and String
     function To_Lower (Item : in Character) return Character;
     function To_Upper (Item : in Character) return Character;
     function To_Basic (Item : in Character) return Character;
     function To_Lower (Item : in String) return String;
     function To_Upper (Item : in String) return String;
     function To_Basic (Item : in String) return String;
   --Classifications of and conversions between Character and ISO 646
     subtype ISO_646 is
       Character range Character'Val(0) .. Character'Val(127);
     function Is_ISO_646 (Item : in Character) return Boolean;
     function Is_ISO_646 (Item : in String)    return Boolean;
     function To_ISO_646 (Item       : in Character;
                          Substitute : in ISO_646 := ' ')
       return ISO_646;
     function To_ISO_646 (Item       : in String;
                          Substitute : in ISO_646 := ' ')
       return String;
   -- The functions Is_Character, Is_String, To_Character, To_String, To_Wide_Character,
   -- and To_Wide_String are obsolescent; see J.14.
```

Paragraphs 14 through 18 were deleted.

```
end Ada.Characters.Handling;
```

In the description below for each function that returns a Boolean result, the effect is described in terms of the conditions under which the value True is returned. If these conditions are not met, then the function returns False.

Each of the following classification functions has a formal Character parameter, Item, and returns a Boolean result.

Is_Control — True if Item is a control character. A *control character* is a character whose position is in one of the ranges 0..31 or 127..159.

Is_Graphic — True if Item is a graphic character. A *graphic character* is a character whose position is in one of the ranges 32..126 or 160..255.

Is_Letter — True if Item is a letter. A *letter* is a character that is in one of the ranges 'A'..'Z' or 'a'..'z', or whose position is in one of the ranges 192..214, 216..246, or 248..255.

Is_Lower — True if Item is a lower-case letter. A *lower-case letter* is a character that is in the range 'a'..'z', or whose position is in one of the ranges 223..246 or 248..255.

Is_Upper — True if Item is an upper-case letter. An *upper-case letter* is a character that is in the range 'A'..'Z' or whose position is in one of the ranges 192..214 or 216.. 222.

Is_Basic — True if Item is a basic letter. A *basic letter* is a character that is in one of the ranges 'A'..'Z' and 'a'..'z', or that is one of the following: 'Æ', 'æ', 'Ð', 'ð', 'Þ', 'þ', or 'ß'.

Is_Digit — True if Item is a decimal digit. A *decimal digit* is a character in the range '0'..'9'.

Is_Decimal_Digit — A renaming of Is_Digit.

Is_Hexadecimal_Digit — True if Item is a hexadecimal digit. A *hexadecimal digit* is a character that is either a decimal digit or that is in one of the ranges 'A' .. 'F' or 'a' .. 'f'.

Is_Alphanumeric 31
True if Item is an alphanumeric character. An *alphanumeric character* is a character that is either a letter or a decimal digit.

Is_Special True if Item is a special graphic character. A *special graphic character* is a graphic character that is not alphanumeric. 32

Is_Line_Terminator 32.1/3
True if Item is a character with position 10 .. 13 (Line_Feed, Line_Tabulation, Form_Feed, Carriage_Return) or 133 (Next_Line).

Is_Mark Never True (no value of type Character has categories Mark, Non-Spacing or Mark, Spacing Combining). 32.2/3

Is_Other_Format 32.3/3
True if Item is a character with position 173 (Soft_Hyphen).

Is_Punctuation_Connector 32.4/3
True if Item is a character with position 95 ('_', known as Low_Line or Underscore).

Is_Space True if Item is a character with position 32 (' ') or 160 (No_Break_Space). 32.5/3

Is_NFKC True if Item can be present in a string normalized to Normalization Form KC (as defined by Definition D121 of The Unicode Standard, version 15.0.0); this includes all characters except those with position 160 (No_Break_Space), 168 ('¨'), 170 ('ª'), 175 ('¯'), 178 ('²'), 179 ('³'), 180 ('´'), 181 ('µ'), 184 (',¸'), 185 ('¹'), 186 ('º'), 188 ('¼'), 189 ('½'), and 190 ('¾'). 32.6/6

Each of the names To_Lower, To_Upper, and To_Basic refers to two functions: one that converts from Character to Character, and the other that converts from String to String. The result of each Character-to-Character function is described below, in terms of the conversion applied to Item, its formal Character parameter. The result of each String-to-String conversion is obtained by applying to each element of the function's String parameter the corresponding Character-to-Character conversion; the result is the null String if the value of the formal parameter is the null String. The lower bound of the result String is 1. 33

To_Lower Returns the corresponding lower-case value for Item if Is_Upper(Item), and returns Item otherwise. 34

To_Upper Returns the corresponding upper-case value for Item if Is_Lower(Item) and Item has an upper-case form, and returns Item otherwise. The lower case letters 'ß' and 'ÿ' do not have upper case forms. 35

To_Basic Returns the letter corresponding to Item but with no diacritical mark, if Item is a letter but not a basic letter; returns Item otherwise. 36

The following set of functions test for membership in the ISO 646 character range, or convert between ISO 646 and Character. 37

Is_ISO_646 The function whose formal parameter, Item, is of type Character returns True if Item is in the subtype ISO_646. 38

Is_ISO_646 The function whose formal parameter, Item, is of type String returns True if Is_ISO_646(Item(I)) is True for each I in Item'Range. 39

To_ISO_646 40
The function whose first formal parameter, Item, is of type Character returns Item if Is_ISO_646(Item), and returns the Substitute ISO_646 character otherwise.

41 To_ISO_646

The function whose first formal parameter, Item, is of type String returns the String whose Range is 1..Item'Length and each of whose elements is given by To_ISO_646 of the corresponding element in Item.

Paragraphs 42 through 49 were deleted.

50 NOTE 1 A basic letter is a letter without a diacritical mark.

51 NOTE 2 Except for the hexadecimal digits, basic letters, and ISO_646 characters, the categories identified in the classification functions form a strict hierarchy:

52 — Control characters

53 — Graphic characters

54 — Alphanumeric characters

55 — Letters

56 — Upper-case letters

57 — Lower-case letters

58 — Decimal digits

59 — Special graphic characters

60/6 NOTE 3 There are certain characters which are defined to be lower case letters by Unicode and are therefore allowed in identifiers, but are not considered lower case letters by Ada.Characters.Handling.

A.3.3 The Package Characters.Latin_1

1 The package Characters.Latin_1 declares constants for characters in ISO 8859-1.

Static Semantics

2 The library package Characters.Latin_1 has the following declaration:

3/5
```
package Ada.Characters.Latin_1
    with Pure is
```

4 *-- Control characters:*

5
```
    NUL                 : constant Character := Character'Val(0);
    SOH                 : constant Character := Character'Val(1);
    STX                 : constant Character := Character'Val(2);
    ETX                 : constant Character := Character'Val(3);
    EOT                 : constant Character := Character'Val(4);
    ENQ                 : constant Character := Character'Val(5);
    ACK                 : constant Character := Character'Val(6);
    BEL                 : constant Character := Character'Val(7);
    BS                  : constant Character := Character'Val(8);
    HT                  : constant Character := Character'Val(9);
    LF                  : constant Character := Character'Val(10);
    VT                  : constant Character := Character'Val(11);
    FF                  : constant Character := Character'Val(12);
    CR                  : constant Character := Character'Val(13);
    SO                  : constant Character := Character'Val(14);
    SI                  : constant Character := Character'Val(15);
```

```
   DLE                    : constant Character := Character'Val(16);
   DC1                    : constant Character := Character'Val(17);
   DC2                    : constant Character := Character'Val(18);
   DC3                    : constant Character := Character'Val(19);
   DC4                    : constant Character := Character'Val(20);
   NAK                    : constant Character := Character'Val(21);
   SYN                    : constant Character := Character'Val(22);
   ETB                    : constant Character := Character'Val(23);
   CAN                    : constant Character := Character'Val(24);
   EM                     : constant Character := Character'Val(25);
   SUB                    : constant Character := Character'Val(26);
   ESC                    : constant Character := Character'Val(27);
   FS                     : constant Character := Character'Val(28);
   GS                     : constant Character := Character'Val(29);
   RS                     : constant Character := Character'Val(30);
   US                     : constant Character := Character'Val(31);

-- ISO 646 graphic characters:

   Space                  : constant Character := ' ';  -- Character'Val(32)
   Exclamation            : constant Character := '!';  -- Character'Val(33)
   Quotation              : constant Character := '"';  -- Character'Val(34)
   Number_Sign            : constant Character := '#';  -- Character'Val(35)
   Dollar_Sign            : constant Character := '$';  -- Character'Val(36)
   Percent_Sign           : constant Character := '%';  -- Character'Val(37)
   Ampersand              : constant Character := '&';  -- Character'Val(38)
   Apostrophe             : constant Character := ''';  -- Character'Val(39)
   Left_Parenthesis       : constant Character := '(';  -- Character'Val(40)
   Right_Parenthesis      : constant Character := ')';  -- Character'Val(41)
   Asterisk               : constant Character := '*';  -- Character'Val(42)
   Plus_Sign              : constant Character := '+';  -- Character'Val(43)
   Comma                  : constant Character := ',';  -- Character'Val(44)
   Hyphen                 : constant Character := '-';  -- Character'Val(45)
   Minus_Sign             : Character renames Hyphen;
   Full_Stop              : constant Character := '.';  -- Character'Val(46)
   Solidus                : constant Character := '/';  -- Character'Val(47)

   -- Decimal digits '0' though '9' are at positions 48 through 57

   Colon                  : constant Character := ':';  -- Character'Val(58)
   Semicolon              : constant Character := ';';  -- Character'Val(59)
   Less_Than_Sign         : constant Character := '<';  -- Character'Val(60)
   Equals_Sign            : constant Character := '=';  -- Character'Val(61)
   Greater_Than_Sign      : constant Character := '>';  -- Character'Val(62)
   Question               : constant Character := '?';  -- Character'Val(63)
   Commercial_At          : constant Character := '@';  -- Character'Val(64)

   -- Letters 'A' through 'Z' are at positions 65 through 90

   Left_Square_Bracket    : constant Character := '[';  -- Character'Val(91)
   Reverse_Solidus        : constant Character := '\';  -- Character'Val(92)
   Right_Square_Bracket   : constant Character := ']';  -- Character'Val(93)
   Circumflex             : constant Character := '^';  -- Character'Val(94)
   Low_Line               : constant Character := '_';  -- Character'Val(95)

   Grave                  : constant Character := '`';  -- Character'Val(96)
   LC_A                   : constant Character := 'a';  -- Character'Val(97)
   LC_B                   : constant Character := 'b';  -- Character'Val(98)
   LC_C                   : constant Character := 'c';  -- Character'Val(99)
   LC_D                   : constant Character := 'd';  -- Character'Val(100)
   LC_E                   : constant Character := 'e';  -- Character'Val(101)
   LC_F                   : constant Character := 'f';  -- Character'Val(102)
   LC_G                   : constant Character := 'g';  -- Character'Val(103)
   LC_H                   : constant Character := 'h';  -- Character'Val(104)
   LC_I                   : constant Character := 'i';  -- Character'Val(105)
   LC_J                   : constant Character := 'j';  -- Character'Val(106)
   LC_K                   : constant Character := 'k';  -- Character'Val(107)
   LC_L                   : constant Character := 'l';  -- Character'Val(108)
   LC_M                   : constant Character := 'm';  -- Character'Val(109)
   LC_N                   : constant Character := 'n';  -- Character'Val(110)
   LC_O                   : constant Character := 'o';  -- Character'Val(111)
```

14
```
   LC_P                : constant Character := 'p';  -- Character'Val(112)
   LC_Q                : constant Character := 'q';  -- Character'Val(113)
   LC_R                : constant Character := 'r';  -- Character'Val(114)
   LC_S                : constant Character := 's';  -- Character'Val(115)
   LC_T                : constant Character := 't';  -- Character'Val(116)
   LC_U                : constant Character := 'u';  -- Character'Val(117)
   LC_V                : constant Character := 'v';  -- Character'Val(118)
   LC_W                : constant Character := 'w';  -- Character'Val(119)
   LC_X                : constant Character := 'x';  -- Character'Val(120)
   LC_Y                : constant Character := 'y';  -- Character'Val(121)
   LC_Z                : constant Character := 'z';  -- Character'Val(122)
   Left_Curly_Bracket  : constant Character := '{';  -- Character'Val(123)
   Vertical_Line       : constant Character := '|';  -- Character'Val(124)
   Right_Curly_Bracket : constant Character := '}';  -- Character'Val(125)
   Tilde               : constant Character := '~';  -- Character'Val(126)
   DEL                 : constant Character := Character'Val(127);
```

15
```
   -- ISO 6429 control characters:
```

16
```
   IS4                 : Character renames FS;
   IS3                 : Character renames GS;
   IS2                 : Character renames RS;
   IS1                 : Character renames US;
```

17
```
   Reserved_128        : constant Character := Character'Val(128);
   Reserved_129        : constant Character := Character'Val(129);
   BPH                 : constant Character := Character'Val(130);
   NBH                 : constant Character := Character'Val(131);
   Reserved_132        : constant Character := Character'Val(132);
   NEL                 : constant Character := Character'Val(133);
   SSA                 : constant Character := Character'Val(134);
   ESA                 : constant Character := Character'Val(135);
   HTS                 : constant Character := Character'Val(136);
   HTJ                 : constant Character := Character'Val(137);
   VTS                 : constant Character := Character'Val(138);
   PLD                 : constant Character := Character'Val(139);
   PLU                 : constant Character := Character'Val(140);
   RI                  : constant Character := Character'Val(141);
   SS2                 : constant Character := Character'Val(142);
   SS3                 : constant Character := Character'Val(143);
```

18
```
   DCS                 : constant Character := Character'Val(144);
   PU1                 : constant Character := Character'Val(145);
   PU2                 : constant Character := Character'Val(146);
   STS                 : constant Character := Character'Val(147);
   CCH                 : constant Character := Character'Val(148);
   MW                  : constant Character := Character'Val(149);
   SPA                 : constant Character := Character'Val(150);
   EPA                 : constant Character := Character'Val(151);
```

19
```
   SOS                 : constant Character := Character'Val(152);
   Reserved_153        : constant Character := Character'Val(153);
   SCI                 : constant Character := Character'Val(154);
   CSI                 : constant Character := Character'Val(155);
   ST                  : constant Character := Character'Val(156);
   OSC                 : constant Character := Character'Val(157);
   PM                  : constant Character := Character'Val(158);
   APC                 : constant Character := Character'Val(159);
```

```
   -- Other graphic characters:                                                    20
   -- Character positions 160 (16#A0#) .. 175 (16#AF#):                           21/3
      No_Break_Space             : constant Character := ' '; --Character'Val(160)
      NBSP                       : Character renames No_Break_Space;
      Inverted_Exclamation       : constant Character := '¡'; --Character'Val(161)
      Cent_Sign                  : constant Character := '¢'; --Character'Val(162)
      Pound_Sign                 : constant Character := '£'; --Character'Val(163)
      Currency_Sign              : constant Character := '¤'; --Character'Val(164)
      Yen_Sign                   : constant Character := '¥'; --Character'Val(165)
      Broken_Bar                 : constant Character := '¦'; --Character'Val(166)
      Section_Sign               : constant Character := '§'; --Character'Val(167)
      Diaeresis                  : constant Character := '¨'; --Character'Val(168)
      Copyright_Sign             : constant Character := '©'; --Character'Val(169)
      Feminine_Ordinal_Indicator : constant Character := 'ª'; --Character'Val(170)
      Left_Angle_Quotation       : constant Character := '«'; --Character'Val(171)
      Not_Sign                   : constant Character := '¬'; --Character'Val(172)
      Soft_Hyphen                : constant Character := Character'Val(173);
      Registered_Trade_Mark_Sign : constant Character := '®'; --Character'Val(174)
      Macron                     : constant Character := '¯'; --Character'Val(175)
   -- Character positions 176 (16#B0#) .. 191 (16#BF#):                           22
      Degree_Sign                : constant Character := '°'; --Character'Val(176)
      Ring_Above                 : Character renames Degree_Sign;
      Plus_Minus_Sign            : constant Character := '±'; --Character'Val(177)
      Superscript_Two            : constant Character := '²'; --Character'Val(178)
      Superscript_Three          : constant Character := '³'; --Character'Val(179)
      Acute                      : constant Character := '´'; --Character'Val(180)
      Micro_Sign                 : constant Character := 'µ'; --Character'Val(181)
      Pilcrow_Sign               : constant Character := '¶'; --Character'Val(182)
      Paragraph_Sign             : Character renames Pilcrow_Sign;
      Middle_Dot                 : constant Character := '·'; --Character'Val(183)
      Cedilla                    : constant Character := '¸'; --Character'Val(184)
      Superscript_One            : constant Character := '¹'; --Character'Val(185)
      Masculine_Ordinal_Indicator: constant Character := 'º'; --Character'Val(186)
      Right_Angle_Quotation      : constant Character := '»'; --Character'Val(187)
      Fraction_One_Quarter       : constant Character := '¼'; --Character'Val(188)
      Fraction_One_Half          : constant Character := '½'; --Character'Val(189)
      Fraction_Three_Quarters    : constant Character := '¾'; --Character'Val(190)
      Inverted_Question          : constant Character := '¿'; --Character'Val(191)
   -- Character positions 192 (16#C0#) .. 207 (16#CF#):                           23
      UC_A_Grave                 : constant Character := 'À'; --Character'Val(192)
      UC_A_Acute                 : constant Character := 'Á'; --Character'Val(193)
      UC_A_Circumflex            : constant Character := 'Â'; --Character'Val(194)
      UC_A_Tilde                 : constant Character := 'Ã'; --Character'Val(195)
      UC_A_Diaeresis             : constant Character := 'Ä'; --Character'Val(196)
      UC_A_Ring                  : constant Character := 'Å'; --Character'Val(197)
      UC_AE_Diphthong            : constant Character := 'Æ'; --Character'Val(198)
      UC_C_Cedilla               : constant Character := 'Ç'; --Character'Val(199)
      UC_E_Grave                 : constant Character := 'È'; --Character'Val(200)
      UC_E_Acute                 : constant Character := 'É'; --Character'Val(201)
      UC_E_Circumflex            : constant Character := 'Ê'; --Character'Val(202)
      UC_E_Diaeresis             : constant Character := 'Ë'; --Character'Val(203)
      UC_I_Grave                 : constant Character := 'Ì'; --Character'Val(204)
      UC_I_Acute                 : constant Character := 'Í'; --Character'Val(205)
      UC_I_Circumflex            : constant Character := 'Î'; --Character'Val(206)
      UC_I_Diaeresis             : constant Character := 'Ï'; --Character'Val(207)
```

24
```
   -- Character positions 208 (16#D0#) .. 223 (16#DF#):
      UC_Icelandic_Eth           : constant Character := 'Ð'; --Character'Val(208)
      UC_N_Tilde                 : constant Character := 'Ñ'; --Character'Val(209)
      UC_O_Grave                 : constant Character := 'Ò'; --Character'Val(210)
      UC_O_Acute                 : constant Character := 'Ó'; --Character'Val(211)
      UC_O_Circumflex            : constant Character := 'Ô'; --Character'Val(212)
      UC_O_Tilde                 : constant Character := 'Õ'; --Character'Val(213)
      UC_O_Diaeresis             : constant Character := 'Ö'; --Character'Val(214)
      Multiplication_Sign        : constant Character := '×'; --Character'Val(215)
      UC_O_Oblique_Stroke        : constant Character := 'Ø'; --Character'Val(216)
      UC_U_Grave                 : constant Character := 'Ù'; --Character'Val(217)
      UC_U_Acute                 : constant Character := 'Ú'; --Character'Val(218)
      UC_U_Circumflex            : constant Character := 'Û'; --Character'Val(219)
      UC_U_Diaeresis             : constant Character := 'Ü'; --Character'Val(220)
      UC_Y_Acute                 : constant Character := 'Ý'; --Character'Val(221)
      UC_Icelandic_Thorn         : constant Character := 'Þ'; --Character'Val(222)
      LC_German_Sharp_S          : constant Character := 'ß'; --Character'Val(223)
```
25
```
   -- Character positions 224 (16#E0#) .. 239 (16#EF#):
      LC_A_Grave                 : constant Character := 'à'; --Character'Val(224)
      LC_A_Acute                 : constant Character := 'á'; --Character'Val(225)
      LC_A_Circumflex            : constant Character := 'â'; --Character'Val(226)
      LC_A_Tilde                 : constant Character := 'ã'; --Character'Val(227)
      LC_A_Diaeresis             : constant Character := 'ä'; --Character'Val(228)
      LC_A_Ring                  : constant Character := 'å'; --Character'Val(229)
      LC_AE_Diphthong            : constant Character := 'æ'; --Character'Val(230)
      LC_C_Cedilla               : constant Character := 'ç'; --Character'Val(231)
      LC_E_Grave                 : constant Character := 'è'; --Character'Val(232)
      LC_E_Acute                 : constant Character := 'é'; --Character'Val(233)
      LC_E_Circumflex            : constant Character := 'ê'; --Character'Val(234)
      LC_E_Diaeresis             : constant Character := 'ë'; --Character'Val(235)
      LC_I_Grave                 : constant Character := 'ì'; --Character'Val(236)
      LC_I_Acute                 : constant Character := 'í'; --Character'Val(237)
      LC_I_Circumflex            : constant Character := 'î'; --Character'Val(238)
      LC_I_Diaeresis             : constant Character := 'ï'; --Character'Val(239)
```
26
```
   -- Character positions 240 (16#F0#) .. 255 (16#FF#):
      LC_Icelandic_Eth           : constant Character := 'ð'; --Character'Val(240)
      LC_N_Tilde                 : constant Character := 'ñ'; --Character'Val(241)
      LC_O_Grave                 : constant Character := 'ò'; --Character'Val(242)
      LC_O_Acute                 : constant Character := 'ó'; --Character'Val(243)
      LC_O_Circumflex            : constant Character := 'ô'; --Character'Val(244)
      LC_O_Tilde                 : constant Character := 'õ'; --Character'Val(245)
      LC_O_Diaeresis             : constant Character := 'ö'; --Character'Val(246)
      Division_Sign              : constant Character := '÷'; --Character'Val(247)
      LC_O_Oblique_Stroke        : constant Character := 'ø'; --Character'Val(248)
      LC_U_Grave                 : constant Character := 'ù'; --Character'Val(249)
      LC_U_Acute                 : constant Character := 'ú'; --Character'Val(250)
      LC_U_Circumflex            : constant Character := 'û'; --Character'Val(251)
      LC_U_Diaeresis             : constant Character := 'ü'; --Character'Val(252)
      LC_Y_Acute                 : constant Character := 'ý'; --Character'Val(253)
      LC_Icelandic_Thorn         : constant Character := 'þ'; --Character'Val(254)
      LC_Y_Diaeresis             : constant Character := 'ÿ'; --Character'Val(255)
end Ada.Characters.Latin_1;
```

Implementation Permissions

27 An implementation may provide additional packages as children of Ada.Characters, to declare names for the symbols of the local character set or other character sets.

A.3.4 The Package Characters.Conversions

Static Semantics

The library package Characters.Conversions has the following declaration: 1/2

```
package Ada.Characters.Conversions                                              2/5
   with Pure is

   function Is_Character (Item : in Wide_Character)      return Boolean;         3/2
   function Is_String    (Item : in Wide_String)         return Boolean;
   function Is_Character (Item : in Wide_Wide_Character) return Boolean;
   function Is_String    (Item : in Wide_Wide_String)    return Boolean;
   function Is_Wide_Character (Item : in Wide_Wide_Character)
      return Boolean;
   function Is_Wide_String    (Item : in Wide_Wide_String)
      return Boolean;

   function To_Wide_Character (Item : in Character) return Wide_Character;      4/2
   function To_Wide_String    (Item : in String)    return Wide_String;
   function To_Wide_Wide_Character (Item : in Character)
      return Wide_Wide_Character;
   function To_Wide_Wide_String    (Item : in String)
      return Wide_Wide_String;
   function To_Wide_Wide_Character (Item : in Wide_Character)
      return Wide_Wide_Character;
   function To_Wide_Wide_String    (Item : in Wide_String)
      return Wide_Wide_String;

   function To_Character (Item       : in Wide_Character;                      5/2
                          Substitute : in Character := ' ')
      return Character;
   function To_String    (Item       : in Wide_String;
                          Substitute : in Character := ' ')
      return String;
   function To_Character (Item :       in Wide_Wide_Character;
                          Substitute : in Character := ' ')
      return Character;
   function To_String    (Item :       in Wide_Wide_String;
                          Substitute : in Character := ' ')
      return String;
   function To_Wide_Character (Item :       in Wide_Wide_Character;
                               Substitute : in Wide_Character := ' ')
      return Wide_Character;
   function To_Wide_String    (Item :       in Wide_Wide_String;
                               Substitute : in Wide_Character := ' ')
      return Wide_String;

end Ada.Characters.Conversions;                                                 6/2
```

The functions in package Characters.Conversions test Wide_Wide_Character or Wide_Character values for membership in Wide_Character or Character, or convert between corresponding characters of Wide_Wide_Character, Wide_Character, and Character. 7/2

```
function Is_Character (Item : in Wide_Character) return Boolean;                8/2
```

Returns True if Wide_Character'Pos(Item) <= Character'Pos(Character'Last). 9/2

```
function Is_Character (Item : in Wide_Wide_Character) return Boolean;           10/2
```

Returns True if Wide_Wide_Character'Pos(Item) <= Character'Pos(Character'Last). 11/2

```
function Is_Wide_Character (Item : in Wide_Wide_Character) return Boolean;      12/2
```

Returns True if Wide_Wide_Character'Pos(Item) <= Wide_Character'Pos(Wide_Character'Last). 13/2

14/2
```
function Is_String (Item : in Wide_String)      return Boolean;
function Is_String (Item : in Wide_Wide_String) return Boolean;
```

15/2 Returns True if Is_Character(Item(I)) is True for each I in Item'Range.

16/2
```
function Is_Wide_String (Item : in Wide_Wide_String) return Boolean;
```

17/2 Returns True if Is_Wide_Character(Item(I)) is True for each I in Item'Range.

18/2
```
function To_Character (Item :       in Wide_Character;
                       Substitute : in Character := ' ') return Character;
function To_Character (Item :       in Wide_Wide_Character;
                       Substitute : in Character := ' ') return Character;
```

19/2 Returns the Character corresponding to Item if Is_Character(Item), and returns the Substitute Character otherwise.

20/2
```
function To_Wide_Character (Item : in Character) return Wide_Character;
```

21/2 Returns the Wide_Character X such that Character'Pos(Item) = Wide_Character'Pos (X).

22/2
```
function To_Wide_Character (Item :       in Wide_Wide_Character;
                            Substitute : in Wide_Character := ' ')
   return Wide_Character;
```

23/2 Returns the Wide_Character corresponding to Item if Is_Wide_Character(Item), and returns the Substitute Wide_Character otherwise.

24/2
```
function To_Wide_Wide_Character (Item : in Character)
   return Wide_Wide_Character;
```

25/2 Returns the Wide_Wide_Character X such that Character'Pos(Item) = Wide_Wide_Character'Pos (X).

26/2
```
function To_Wide_Wide_Character (Item : in Wide_Character)
   return Wide_Wide_Character;
```

27/2 Returns the Wide_Wide_Character X such that Wide_Character'Pos(Item) = Wide_Wide_Character'Pos (X).

28/2
```
function To_String (Item :       in Wide_String;
                    Substitute : in Character := ' ') return String;
function To_String (Item :       in Wide_Wide_String;
                    Substitute : in Character := ' ') return String;
```

29/2 Returns the String whose range is 1..Item'Length and each of whose elements is given by To_Character of the corresponding element in Item.

30/2
```
function To_Wide_String (Item : in String) return Wide_String;
```

31/2 Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item.

32/2
```
function To_Wide_String (Item :       in Wide_Wide_String;
                         Substitute : in Wide_Character := ' ')
   return Wide_String;
```

33/2 Returns the Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Character of the corresponding element in Item with the given Substitute Wide_Character.

```
function To_Wide_Wide_String (Item : in String) return Wide_Wide_String;
function To_Wide_Wide_String (Item : in Wide_String)
   return Wide_Wide_String;
```
34/2

Returns the Wide_Wide_String whose range is 1..Item'Length and each of whose elements is given by To_Wide_Wide_Character of the corresponding element in Item. 35/2

A.3.5 The Package Wide_Characters.Handling

The package Wide_Characters.Handling provides operations for classifying Wide_Characters and case folding for Wide_Characters. 1/3

Static Semantics

The library package Wide_Characters.Handling has the following declaration: 2/3

```
package Ada.Wide_Characters.Handling
   with Pure is
   function Character_Set_Version return String;
   function Is_Control (Item : Wide_Character) return Boolean;
   function Is_Letter (Item : Wide_Character) return Boolean;
   function Is_Lower (Item : Wide_Character) return Boolean;
   function Is_Upper (Item : Wide_Character) return Boolean;
   function Is_Basic (Item : Wide_Character) return Boolean;
   function Is_Digit (Item : Wide_Character) return Boolean;
   function Is_Decimal_Digit (Item : Wide_Character) return Boolean
      renames Is_Digit;
   function Is_Hexadecimal_Digit (Item : Wide_Character) return Boolean;
   function Is_Alphanumeric (Item : Wide_Character) return Boolean;
   function Is_Special (Item : Wide_Character) return Boolean;
   function Is_Line_Terminator (Item : Wide_Character) return Boolean;
   function Is_Mark (Item : Wide_Character) return Boolean;
   function Is_Other_Format (Item : Wide_Character) return Boolean;
   function Is_Punctuation_Connector (Item : Wide_Character) return Boolean;
   function Is_Space (Item : Wide_Character) return Boolean;
   function Is_NFKC (Item : Wide_Character) return Boolean;
   function Is_Graphic (Item : Wide_Character) return Boolean;
   function To_Lower (Item : Wide_Character) return Wide_Character;
   function To_Upper (Item : Wide_Character) return Wide_Character;
   function To_Basic (Item : Wide_Character) return Wide_Character;
   function To_Lower (Item : Wide_Character) return Wide_String;
   function To_Upper (Item : Wide_String) return Wide_String;
   function To_Basic (Item : Wide_String) return Wide_String;
end Ada.Wide_Characters.Handling;
```
3/5 4/3 5/3 6/3 7/3 8/3 8.1/5 9/3 10/3 11/3 12/3 13/3 14/3 15/3 16/3 17/3 18/3 18.1/5 19/3 20/3 20.1/5 21/3 21.1/5 22/3

The subprograms defined in Wide_Characters.Handling are locale independent. 23/3

```
function Character_Set_Version return String;
```
24/3

Returns an implementation-defined identifier that identifies the version of the character set standard that is used for categorizing characters by the implementation. 25/3

26/3 `function Is_Control (Item : Wide_Character) return Boolean;`

27/3 Returns True if the Wide_Character designated by Item is categorized as other_control; otherwise returns False.

28/3 `function Is_Letter (Item : Wide_Character) return Boolean;`

29/3 Returns True if the Wide_Character designated by Item is categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, or number_letter; otherwise returns False.

30/3 `function Is_Lower (Item : Wide_Character) return Boolean;`

31/3 Returns True if the Wide_Character designated by Item is categorized as letter_lowercase; otherwise returns False.

32/3 `function Is_Upper (Item : Wide_Character) return Boolean;`

33/3 Returns True if the Wide_Character designated by Item is categorized as letter_uppercase; otherwise returns False.

33.1/5 `function Is_Basic (Item : Wide_Character) return Boolean;`

33.2/6 Returns True if the Wide_Character designated by Item has no Decomposition Mapping in the code charts of The Unicode Standard, version 15.0.0; otherwise returns False.

34/3 `function Is_Digit (Item : Wide_Character) return Boolean;`

35/3 Returns True if the Wide_Character designated by Item is categorized as number_decimal; otherwise returns False.

36/3 `function Is_Hexadecimal_Digit (Item : Wide_Character) return Boolean;`

37/3 Returns True if the Wide_Character designated by Item is categorized as number_decimal, or is in the range 'A' .. 'F' or 'a' .. 'f'; otherwise returns False.

38/3 `function Is_Alphanumeric (Item : Wide_Character) return Boolean;`

39/3 Returns True if the Wide_Character designated by Item is categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, number_letter, or number_decimal; otherwise returns False.

40/3 `function Is_Special (Item : Wide_Character) return Boolean;`

41/3 Returns True if the Wide_Character designated by Item is categorized as graphic_character, but not categorized as letter_uppercase, letter_lowercase, letter_titlecase, letter_modifier, letter_other, number_letter, or number_decimal; otherwise returns False.

42/3 `function Is_Line_Terminator (Item : Wide_Character) return Boolean;`

43/3 Returns True if the Wide_Character designated by Item is categorized as separator_line or separator_paragraph, or if Item is a conventional line terminator character (Line_Feed, Line_Tabulation, Form_Feed, Carriage_Return, Next_Line); otherwise returns False.

44/3 `function Is_Mark (Item : Wide_Character) return Boolean;`

45/3 Returns True if the Wide_Character designated by Item is categorized as mark_non_spacing or mark_spacing_combining; otherwise returns False.

```
function Is_Other_Format (Item : Wide_Character) return Boolean;
```
46/3

Returns True if the Wide_Character designated by Item is categorized as other_format; otherwise returns False. 47/3

```
function Is_Punctuation_Connector (Item : Wide_Character) return Boolean;
```
48/3

Returns True if the Wide_Character designated by Item is categorized as punctuation_connector; otherwise returns False. 49/3

```
function Is_Space (Item : Wide_Character) return Boolean;
```
50/3

Returns True if the Wide_Character designated by Item is categorized as separator_space; otherwise returns False. 51/3

```
function Is_NFKC (Item : Wide_Character) return Boolean;
```
51.1/5

Returns True if the Wide_Character designated by Item can be present in a string normalized to Normalization Form KC (as defined by Definition D121 of The Unicode Standard, version 15.0.0), otherwise returns False. 51.2/6

```
function Is_Graphic (Item : Wide_Character) return Boolean;
```
52/3

Returns True if the Wide_Character designated by Item is categorized as graphic_character; otherwise returns False. 53/3

```
function To_Lower (Item : Wide_Character) return Wide_Character;
```
54/3

Returns the Simple Lowercase Mapping as defined by the Unicode Character Database of the Wide_Character designated by Item. 55/6

```
function To_Lower (Item : Wide_String) return Wide_String;
```
56/3

Returns the result of applying the To_Lower conversion to each Wide_Character element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String. The lower bound of the result Wide_String is 1. 57/3

```
function To_Upper (Item : Wide_Character) return Wide_Character;
```
58/3

Returns the Simple Uppercase Mapping as defined by the Unicode Character Database of the Wide_Character designated by Item. 59/6

```
function To_Upper (Item : Wide_String) return Wide_String;
```
60/3

Returns the result of applying the To_Upper conversion to each Wide_Character element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String. The lower bound of the result Wide_String is 1. 61/3

```
function To_Basic (Item : Wide_Character) return Wide_Character;
```
61.1/5

Returns the Wide_Character whose code point is given by the first value of its Decomposition Mapping in the code charts of The Unicode Standard, version 15.0.0 if any; returns Item otherwise. 61.2/6

```
function To_Basic (Item : Wide_String) return Wide_String;
```
61.3/5

Returns the result of applying the To_Basic conversion to each Wide_Character element of the Wide_String designated by Item. The result is the null Wide_String if the value of the formal parameter is the null Wide_String. The lower bound of the result Wide_String is 1. 61.4/5

Implementation Advice

62/3 The string returned by Character_Set_Version should include either “10646:” or “Unicode”.

63/6 NOTE 1 The results returned by these functions can depend on which particular version of Unicode or ISO/IEC 10646 is supported by the implementation (see 2.1).

64/5 NOTE 2 The case insensitive equality comparison routines provided in A.4.10 are also available for wide strings (see A.4.7).

A.3.6 The Package Wide_Wide_Characters.Handling

1/3 The package Wide_Wide_Characters.Handling has the same contents as Wide_Characters.Handling except that each occurrence of Wide_Character is replaced by Wide_Wide_Character, and each occurrence of Wide_String is replaced by Wide_Wide_String.

A.4 String Handling

1/3 This subclause presents the specifications of the package Strings and several child packages, which provide facilities for dealing with string data. Fixed-length, bounded-length, and unbounded-length strings are supported, for String, Wide_String, and Wide_Wide_String. The string-handling subprograms include searches for pattern strings and for characters in program-specified sets, translation (via a character-to-character mapping), and transformation (replacing, inserting, overwriting, and deleting of substrings).

A.4.1 The Package Strings

1 The package Strings provides declarations common to the string handling packages.

Static Semantics

2 The library package Strings has the following declaration:

3/5
```
package Ada.Strings
   with Pure is
```

4/2
```
   Space      : constant Character      := ' ';
   Wide_Space : constant Wide_Character := ' ';
   Wide_Wide_Space : constant Wide_Wide_Character := ' ';
```

5
```
   Length_Error, Pattern_Error, Index_Error, Translation_Error : exception;
```

6
```
   type Alignment  is (Left, Right, Center);
   type Truncation is (Left, Right, Error);
   type Membership is (Inside, Outside);
   type Direction  is (Forward, Backward);
   type Trim_End   is (Left, Right, Both);
end Ada.Strings;
```

A.4.2 The Package Strings.Maps

1/5 The package Strings.Maps defines the types, operations, and other entities necessary for character sets and character-to-character mappings.

Static Semantics

2 The library package Strings.Maps has the following declaration:

3/5
```
package Ada.Strings.Maps
   with Pure is
```

```
   -- Representation for a set of character values:                                    4/5
   type Character_Set is private
      with Preelaborable_Initialization;

   Null_Set : constant Character_Set;                                                  5

   type Character_Range is                                                             6
     record
        Low  : Character;
        High : Character;
     end record;
   -- Represents Character range Low..High

   type Character_Ranges is array (Positive range <>) of Character_Range;              7

   function To_Set    (Ranges : in Character_Ranges)return Character_Set;              8

   function To_Set    (Span   : in Character_Range)return Character_Set;               9

   function To_Ranges (Set    : in Character_Set)  return Character_Ranges;            10

   function "="   (Left, Right : in Character_Set) return Boolean;                     11

   function "not" (Right : in Character_Set)       return Character_Set;               12
   function "and" (Left, Right : in Character_Set) return Character_Set;
   function "or"  (Left, Right : in Character_Set) return Character_Set;
   function "xor" (Left, Right : in Character_Set) return Character_Set;
   function "-"   (Left, Right : in Character_Set) return Character_Set;

   function Is_In (Element : in Character;                                             13
                   Set     : in Character_Set)
      return Boolean;

   function Is_Subset (Elements : in Character_Set;                                    14
                       Set      : in Character_Set)
      return Boolean;

   function "<=" (Left  : in Character_Set;                                            15
                  Right : in Character_Set)
      return Boolean renames Is_Subset;

   -- Alternative representation for a set of character values:                        16
   subtype Character_Sequence is String;

   function To_Set (Sequence  : in Character_Sequence)return Character_Set;            17

   function To_Set (Singleton : in Character)     return Character_Set;                18

   function To_Sequence (Set  : in Character_Set) return Character_Sequence;           19

   -- Representation for a character to character mapping:                            20/5
   type Character_Mapping is private
      with Preelaborable_Initialization;

   function Value (Map     : in Character_Mapping;                                     21
                   Element : in Character)
      return Character;

   Identity : constant Character_Mapping;                                              22

   function To_Mapping (From, To : in Character_Sequence)                              23
      return Character_Mapping;

   function To_Domain (Map : in Character_Mapping)                                     24
      return Character_Sequence;
   function To_Range  (Map : in Character_Mapping)
      return Character_Sequence;

   type Character_Mapping_Function is                                                  25
      access function (From : in Character) return Character;

private                                                                                26
   ... -- not specified by the language
end Ada.Strings.Maps;
```

An object of type Character_Set represents a set of characters. 27

Null_Set represents the set containing no characters. 28

29 An object Obj of type Character_Range represents the set of characters in the range Obj.Low .. Obj.High.

30 An object Obj of type Character_Ranges represents the union of the sets corresponding to Obj(I) for I in Obj'Range.

31
```
function To_Set (Ranges : in Character_Ranges) return Character_Set;
```

32/3 If Ranges'Length=0 then Null_Set is returned; otherwise, the returned value represents the set corresponding to Ranges.

33
```
function To_Set (Span : in Character_Range) return Character_Set;
```

34 The returned value represents the set containing each character in Span.

35
```
function To_Ranges (Set : in Character_Set) return Character_Ranges;
```

36/3 If Set = Null_Set, then an empty Character_Ranges array is returned; otherwise, the shortest array of contiguous ranges of Character values in Set, in increasing order of Low, is returned.

37
```
function "=" (Left, Right : in Character_Set) return Boolean;
```

38 The function "=" returns True if Left and Right represent identical sets, and False otherwise.

39 Each of the logical operators "**not**", "**and**", "**or**", and "**xor**" returns a Character_Set value that represents the set obtained by applying the corresponding operation to the set(s) represented by the parameter(s) of the operator. "–"(Left, Right) is equivalent to "and"(Left, "not"(Right)).

40
```
function Is_In (Element : in Character;
                Set     : in Character_Set);
   return Boolean;
```

41 Is_In returns True if Element is in Set, and False otherwise.

42
```
function Is_Subset (Elements : in Character_Set;
                    Set      : in Character_Set)
   return Boolean;
```

43 Is_Subset returns True if Elements is a subset of Set, and False otherwise.

44
```
subtype Character_Sequence is String;
```

45 The Character_Sequence subtype is used to portray a set of character values and also to identify the domain and range of a character mapping.

46
```
function To_Set (Sequence  : in Character_Sequence) return Character_Set;
function To_Set (Singleton : in Character)          return Character_Set;
```

47 Sequence portrays the set of character values that it explicitly contains (ignoring duplicates). Singleton portrays the set comprising a single Character. Each of the To_Set functions returns a Character_Set value that represents the set portrayed by Sequence or Singleton.

48
```
function To_Sequence (Set : in Character_Set) return Character_Sequence;
```

49 The function To_Sequence returns a Character_Sequence value containing each of the characters in the set represented by Set, in ascending order with no duplicates.

50
```
type Character_Mapping is private;
```

51 An object of type Character_Mapping represents a Character-to-Character mapping.

```
function Value (Map     : in Character_Mapping;
                Element : in Character)
   return Character;
```

52

The function Value returns the Character value to which Element maps with respect to the mapping represented by Map. 53

A character C *matches* a pattern character P with respect to a given Character_Mapping value Map if Value(Map, C) = P. A string S *matches* a pattern string P with respect to a given Character_Mapping if their lengths are the same and if each character in S matches its corresponding character in the pattern string P. 54

String handling subprograms that deal with character mappings have parameters whose type is Character_Mapping. 55

```
Identity : constant Character_Mapping;
```

56

Identity maps each Character to itself. 57

```
function To_Mapping (From, To : in Character_Sequence)
    return Character_Mapping;
```

58

To_Mapping produces a Character_Mapping such that each element of From maps to the corresponding element of To, and each other character maps to itself. If From'Length /= To'Length, or if some character is repeated in From, then Translation_Error is propagated. 59

```
function To_Domain (Map : in Character_Mapping) return Character_Sequence;
```

60

To_Domain returns the shortest Character_Sequence value D such that each character not in D maps to itself, and such that the characters in D are in ascending order. The lower bound of D is 1. 61

```
function To_Range  (Map : in Character_Mapping) return Character_Sequence;
```

62

To_Range returns the Character_Sequence value R, such that if D = To_Domain(Map), then R has the same bounds as D, and D(I) maps to R(I) for each I in D'Range. 63/1

An object F of type Character_Mapping_Function maps a Character value C to the Character value F.**all**(C), which is said to *match* C with respect to mapping function F. 64

NOTE 1 Character_Mapping and Character_Mapping_Function are used both for character equivalence mappings in the search subprograms (such as for case insensitivity) and as transformational mappings in the Translate subprograms. 65

NOTE 2 To_Domain(Identity) and To_Range(Identity) each returns the null string. 66

Examples

Example of use of Strings.Maps.To_Mapping: 67/5

To_Mapping("ABCD", "ZZAB") returns a Character_Mapping that maps 'A' and 'B' to 'Z', 'C' to 'A', 'D' to 'B', and each other Character to itself. 68

A.4.3 Fixed-Length String Handling

1 The language-defined package Strings.Fixed provides string-handling subprograms for fixed-length strings; that is, for values of type Standard.String. Several of these subprograms are procedures that modify the contents of a String that is passed as an **out** or an **in out** parameter; each has additional parameters to control the effect when the logical length of the result differs from the parameter's length.

2 For each function that returns a String, the lower bound of the returned value is 1.

3 The basic model embodied in the package is that a fixed-length string comprises significant characters and possibly padding (with space characters) on either or both ends. When a shorter string is copied to a longer string, padding is inserted, and when a longer string is copied to a shorter one, padding is stripped. The Move procedure in Strings.Fixed, which takes a String as an **out** parameter, allows the programmer to control these effects. Similar control is provided by the string transformation procedures.

Static Semantics

4 The library package Strings.Fixed has the following declaration:

5/5
```
with Ada.Strings.Maps;
package Ada.Strings.Fixed
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

6
```
   -- "Copy" procedure for strings of possibly different lengths
```

7
```
   procedure Move (Source  : in  String;
                   Target  : out String;
                   Drop    : in  Truncation := Error;
                   Justify : in  Alignment  := Left;
                   Pad     : in  Character  := Space);
```

8
```
   -- Search subprograms
```

8.1/2
```
   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping := Maps.Identity)
      return Natural;
```

8.2/6
```
   function Index (Source  : in String;
                   Pattern : in String;
                   From    : in Positive;
                   Going   : in Direction := Forward;
                   Mapping : in not null Maps.Character_Mapping_Function)
      return Natural;
```

9
```
   function Index (Source  : in String;
                   Pattern : in String;
                   Going   : in Direction := Forward;
                   Mapping : in Maps.Character_Mapping
                                := Maps.Identity)
      return Natural;
```

10/6
```
   function Index (Source  : in String;
                   Pattern : in String;
                   Going   : in Direction := Forward;
                   Mapping : in not null Maps.Character_Mapping_Function)
      return Natural;
```

10.1/2
```
   function Index (Source : in String;
                   Set    : in Maps.Character_Set;
                   From   : in Positive;
                   Test   : in Membership := Inside;
                   Going  : in Direction := Forward)
      return Natural;
```

```
   function Index (Source : in String;                                          11
                   Set    : in Maps.Character_Set;
                   Test   : in Membership := Inside;
                   Going  : in Direction  := Forward)
      return Natural;
   function Index_Non_Blank (Source : in String;                                11.1/2
                             From   : in Positive;
                             Going  : in Direction := Forward)
      return Natural;
   function Index_Non_Blank (Source : in String;                                12
                             Going  : in Direction := Forward)
      return Natural;
   function Count (Source  : in String;                                         13
                   Pattern : in String;
                   Mapping : in Maps.Character_Mapping
                                := Maps.Identity)
      return Natural;
   function Count (Source  : in String;                                         14/6
                   Pattern : in String;
                   Mapping : in not null Maps.Character_Mapping_Function)
      return Natural;
   function Count (Source  : in String;                                         15
                   Set     : in Maps.Character_Set)
      return Natural;
   procedure Find_Token (Source : in String;                                    15.1/3
                         Set    : in Maps.Character_Set;
                         From   : in Positive;
                         Test   : in Membership;
                         First  : out Positive;
                         Last   : out Natural);
   procedure Find_Token (Source : in String;                                    16
                         Set    : in Maps.Character_Set;
                         Test   : in Membership;
                         First  : out Positive;
                         Last   : out Natural);
-- String translation subprograms                                              17
   function Translate (Source  : in String;                                     18
                       Mapping : in Maps.Character_Mapping)
      return String;
   procedure Translate (Source  : in out String;                                19
                        Mapping : in Maps.Character_Mapping);
   function Translate                                                           20/6
     (Source  : in String;
      Mapping : in not null Maps.Character_Mapping_Function)
      return String;
   procedure Translate                                                          21/6
     (Source  : in out String;
      Mapping : in not null Maps.Character_Mapping_Function);
-- String transformation subprograms                                           22
   function Replace_Slice (Source   : in String;                                23
                           Low      : in Positive;
                           High     : in Natural;
                           By       : in String)
      return String;
```

```
24        procedure Replace_Slice (Source   : in out String;
                                   Low      : in Positive;
                                   High     : in Natural;
                                   By       : in String;
                                   Drop     : in Truncation := Error;
                                   Justify  : in Alignment  := Left;
                                   Pad      : in Character  := Space);
25        function Insert (Source   : in String;
                           Before   : in Positive;
                           New_Item : in String)
             return String;
26        procedure Insert (Source   : in out String;
                            Before   : in Positive;
                            New_Item : in String;
                            Drop     : in Truncation := Error);
27        function Overwrite (Source   : in String;
                              Position : in Positive;
                              New_Item : in String)
             return String;
28        procedure Overwrite (Source   : in out String;
                               Position : in Positive;
                               New_Item : in String;
                               Drop     : in Truncation := Right);
29        function Delete (Source  : in String;
                           From    : in Positive;
                           Through : in Natural)
             return String;
30        procedure Delete (Source  : in out String;
                            From    : in Positive;
                            Through : in Natural;
                            Justify : in Alignment := Left;
                            Pad     : in Character := Space);
31      --String selector subprograms
          function Trim (Source : in String;
                         Side   : in Trim_End)
             return String;
32        procedure Trim (Source  : in out String;
                          Side    : in Trim_End;
                          Justify : in Alignment := Left;
                          Pad     : in Character := Space);
33        function Trim (Source : in String;
                         Left   : in Maps.Character_Set;
                         Right  : in Maps.Character_Set)
             return String;
34        procedure Trim (Source  : in out String;
                          Left    : in Maps.Character_Set;
                          Right   : in Maps.Character_Set;
                          Justify : in Alignment := Strings.Left;
                          Pad     : in Character := Space);
35        function Head (Source : in String;
                         Count  : in Natural;
                         Pad    : in Character := Space)
             return String;
36        procedure Head (Source  : in out String;
                          Count   : in Natural;
                          Justify : in Alignment := Left;
                          Pad     : in Character := Space);
37        function Tail (Source : in String;
                         Count  : in Natural;
                         Pad    : in Character := Space)
             return String;
```

```
   procedure Tail (Source  : in out String;
                   Count   : in Natural;
                   Justify : in Alignment := Left;
                   Pad     : in Character := Space);
```

```
   --String constructor functions
```

```
   function "*" (Left  : in Natural;
                 Right : in Character) return String;
```

```
   function "*" (Left  : in Natural;
                 Right : in String) return String;
```

```
end Ada.Strings.Fixed;
```

The effects of the above subprograms are as follows.

```
procedure Move (Source  : in  String;
                Target  : out String;
                Drop    : in  Truncation := Error;
                Justify : in  Alignment  := Left;
                Pad     : in  Character  := Space);
```

The Move procedure copies characters from Source to Target. If Source has the same length as Target, then the effect is to assign Source to Target. If Source is shorter than Target, then:

- If Justify=Left, then Source is copied into the first Source'Length characters of Target.
- If Justify=Right, then Source is copied into the last Source'Length characters of Target.
- If Justify=Center, then Source is copied into the middle Source'Length characters of Target. In this case, if the difference in length between Target and Source is odd, then the extra Pad character is on the right.
- Pad is copied to each Target character not otherwise assigned.

If Source is longer than Target, then the effect is based on Drop.

- If Drop=Left, then the rightmost Target'Length characters of Source are copied into Target.
- If Drop=Right, then the leftmost Target'Length characters of Source are copied into Target.
- If Drop=Error, then the effect depends on the value of the Justify parameter and also on whether any characters in Source other than Pad would fail to be copied:
  - If Justify=Left, and if each of the rightmost Source'Length-Target'Length characters in Source is Pad, then the leftmost Target'Length characters of Source are copied to Target.
  - If Justify=Right, and if each of the leftmost Source'Length-Target'Length characters in Source is Pad, then the rightmost Target'Length characters of Source are copied to Target.
  - Otherwise, Length_Error is propagated.

56.1/6
```
function Index (Source  : in String;
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping := Maps.Identity)
   return Natural;

function Index (Source  : in String;
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in not null Maps.Character_Mapping_Function)
   return Natural;
```

56.2/3 Each Index function searches, starting from From, for a slice of Source, with length Pattern'Length, that matches Pattern with respect to Mapping; the parameter Going indicates the direction of the lookup. If Source is the null string, Index returns 0; otherwise, if From is not in Source'Range, then Index_Error is propagated. If Going = Forward, then Index returns the smallest index I which is greater than or equal to From such that the slice of Source starting at I matches Pattern. If Going = Backward, then Index returns the largest index I such that the slice of Source starting at I matches Pattern and has an upper bound less than or equal to From. If there is no such slice, then 0 is returned. If Pattern is the null string, then Pattern_Error is propagated.

57/6
```
function Index (Source  : in String;
                Pattern : in String;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping
                             := Maps.Identity)
   return Natural;

function Index (Source  : in String;
                Pattern : in String;
                Going   : in Direction := Forward;
                Mapping : in not null Maps.Character_Mapping_Function)
   return Natural;
```

58/2 If Going = Forward, returns

58.1/2
```
Index (Source, Pattern, Source'First, Forward, Mapping);
```

58.2/3 otherwise, returns

58.3/2
```
Index (Source, Pattern, Source'Last, Backward, Mapping);
```

58.4/2
```
function Index (Source : in String;
                Set    : in Maps.Character_Set;
                From   : in Positive;
                Test   : in Membership := Inside;
                Going  : in Direction := Forward)
   return Natural;
```

58.5/3 Index searches for the first or last occurrence of any of a set of characters (when Test=Inside), or any of the complement of a set of characters (when Test=Outside). If Source is the null string, Index returns 0; otherwise, if From is not in Source'Range, then Index_Error is propagated. Otherwise, it returns the smallest index I >= From (if Going=Forward) or the largest index I <= From (if Going=Backward) such that Source(I) satisfies the Test condition with respect to Set; it returns 0 if there is no such Character in Source.

```
function Index (Source : in String;                                    59
                Set    : in Maps.Character_Set;
                Test   : in Membership := Inside;
                Going  : in Direction  := Forward)
   return Natural;
```

If Going = Forward, returns 60/2

```
   Index (Source, Set, Source'First, Test, Forward);                   60.1/2
```

otherwise, returns 60.2/3

```
   Index (Source, Set, Source'Last, Test, Backward);                   60.3/2
```

```
function Index_Non_Blank (Source : in String;                          60.4/2
                          From   : in Positive;
                          Going  : in Direction := Forward)
   return Natural;
```

Returns Index (Source, Maps.To_Set(Space), From, Outside, Going); 60.5/2

```
function Index_Non_Blank (Source : in String;                          61
                          Going  : in Direction := Forward)
   return Natural;
```

Returns Index(Source, Maps.To_Set(Space), Outside, Going) 62

```
function Count (Source  : in String;                                   63/6
                Pattern : in String;
                Mapping : in Maps.Character_Mapping
                             := Maps.Identity)
   return Natural;

function Count (Source  : in String;
                Pattern : in String;
                Mapping : in not null Maps.Character_Mapping_Function)
   return Natural;
```

Returns the maximum number of nonoverlapping slices of Source that match Pattern with respect to Mapping. If Pattern is the null string then Pattern_Error is propagated. 64

```
function Count (Source  : in String;                                   65
                Set     : in Maps.Character_Set)
   return Natural;
```

Returns the number of occurrences in Source of characters that are in Set. 66

```
procedure Find_Token (Source : in String;                              66.1/3
                      Set    : in Maps.Character_Set;
                      From   : in Positive;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

If Source is not the null string and From is not in Source'Range, then Index_Error is raised. Otherwise, First is set to the index of the first character in Source(From .. Source'Last) that satisfies the Test condition. Last is set to the largest index such that all characters in Source(First .. Last) satisfy the Test condition. If no characters in Source(From .. Source'Last) satisfy the Test condition, First is set to From, and Last is set to 0. 66.2/3

```
procedure Find_Token (Source : in String;                              67
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

Equivalent to Find_Token (Source, Set, Source'First, Test, First, Last). 68/3

69/6
```
function Translate (Source  : in String;
                    Mapping : in Maps.Character_Mapping)
   return String;

function Translate
  (Source  : in String;
   Mapping : in not null Maps.Character_Mapping_Function)
   return String;
```

70 Returns the string S whose length is Source'Length and such that S(I) is the character to which Mapping maps the corresponding element of Source, for I in 1..Source'Length.

71/6
```
procedure Translate (Source  : in out String;
                     Mapping : in Maps.Character_Mapping);

procedure Translate
  (Source  : in out String;
   Mapping : in not null Maps.Character_Mapping_Function);
```

72 Equivalent to Source := Translate(Source, Mapping).

73
```
function Replace_Slice (Source   : in String;
                        Low      : in Positive;
                        High     : in Natural;
                        By       : in String)
   return String;
```

74/1 If Low > Source'Last+1, or High < Source'First–1, then Index_Error is propagated. Otherwise:

74.1/1 - If High >= Low, then the returned string comprises Source(Source'First..Low–1) & By & Source(High+1..Source'Last), but with lower bound 1.

74.2/1 - If High < Low, then the returned string is Insert(Source, Before=>Low, New_Item=>By).

75
```
procedure Replace_Slice (Source   : in out String;
                         Low      : in Positive;
                         High     : in Natural;
                         By       : in String;
                         Drop     : in Truncation := Error;
                         Justify  : in Alignment  := Left;
                         Pad      : in Character  := Space);
```

76 Equivalent to Move(Replace_Slice(Source, Low, High, By), Source, Drop, Justify, Pad).

77
```
function Insert (Source   : in String;
                 Before   : in Positive;
                 New_Item : in String)
   return String;
```

78/3 Propagates Index_Error if Before is not in Source'First .. Source'Last+1; otherwise, returns Source(Source'First..Before–1) & New_Item & Source(Before..Source'Last), but with lower bound 1.

79
```
procedure Insert (Source   : in out String;
                  Before   : in Positive;
                  New_Item : in String;
                  Drop     : in Truncation := Error);
```

80 Equivalent to Move(Insert(Source, Before, New_Item), Source, Drop).

```
function Overwrite (Source   : in String;                                    81
                    Position : in Positive;
                    New_Item : in String)
   return String;
```

Propagates Index_Error if Position is not in Source'First .. Source'Last+1; otherwise, returns the string obtained from Source by consecutively replacing characters starting at Position with corresponding characters from New_Item. If the end of Source is reached before the characters in New_Item are exhausted, the remaining characters from New_Item are appended to the string. 82/3

```
procedure Overwrite (Source   : in out String;                               83
                     Position : in Positive;
                     New_Item : in String;
                     Drop     : in Truncation := Right);
```

Equivalent to Move(Overwrite(Source, Position, New_Item), Source, Drop). 84

```
function Delete (Source  : in String;                                        85
                 From    : in Positive;
                 Through : in Natural)
   return String;
```

If From <= Through, the returned string is Replace_Slice(Source, From, Through, ""); otherwise, it is Source with lower bound 1. 86/3

```
procedure Delete (Source  : in out String;                                   87
                  From    : in Positive;
                  Through : in Natural;
                  Justify : in Alignment := Left;
                  Pad     : in Character := Space);
```

Equivalent to Move(Delete(Source, From, Through), Source, Justify => Justify, Pad => Pad). 88

```
function Trim (Source : in String;                                           89
               Side   : in Trim_End)
  return String;
```

Returns the string obtained by removing from Source all leading Space characters (if Side = Left), all trailing Space characters (if Side = Right), or all leading and trailing Space characters (if Side = Both). 90

```
procedure Trim (Source  : in out String;                                     91
                Side    : in Trim_End;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

Equivalent to Move(Trim(Source, Side), Source, Justify=>Justify, Pad=>Pad). 92

```
function Trim (Source : in String;                                           93
               Left   : in Maps.Character_Set;
               Right  : in Maps.Character_Set)
   return String;
```

Returns the string obtained by removing from Source all leading characters in Left and all trailing characters in Right. 94

```
procedure Trim (Source  : in out String;                                     95
                Left    : in Maps.Character_Set;
                Right   : in Maps.Character_Set;
                Justify : in Alignment := Strings.Left;
                Pad     : in Character := Space);
```

Equivalent to Move(Trim(Source, Left, Right), Source, Justify => Justify, Pad=>Pad). 96

97
```
function Head (Source : in String;
               Count  : in Natural;
               Pad    : in Character := Space)
   return String;
```

98/3 Returns a string of length Count. If Count <= Source'Length, the string comprises the first Count characters of Source. Otherwise, its contents are Source concatenated with Count–Source'Length Pad characters.

99
```
procedure Head (Source  : in out String;
                Count   : in Natural;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

100 Equivalent to Move(Head(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad).

101
```
function Tail (Source : in String;
               Count  : in Natural;
               Pad    : in Character := Space)
   return String;
```

102/3 Returns a string of length Count. If Count <= Source'Length, the string comprises the last Count characters of Source. Otherwise, its contents are Count-Source'Length Pad characters concatenated with Source.

103
```
procedure Tail (Source  : in out String;
                Count   : in Natural;
                Justify : in Alignment := Left;
                Pad     : in Character := Space);
```

104 Equivalent to Move(Tail(Source, Count, Pad), Source, Drop=>Error, Justify=>Justify, Pad=>Pad).

105
```
function "*" (Left  : in Natural;
              Right : in Character) return String;

function "*" (Left  : in Natural;
              Right : in String) return String;
```

106/1 These functions replicate a character or string a specified number of times. The first function returns a string whose length is Left and each of whose elements is Right. The second function returns a string whose length is Left*Right'Length and whose value is the null string if Left = 0 and otherwise is (Left–1)*Right & Right with lower bound 1.

107/5 NOTE 1 In the Index and Count functions taking Pattern and Mapping parameters, for there to be a match, the actual String parameter passed to Pattern can contain only characters occurring as target characters of the mapping.

108 NOTE 2 In the Insert subprograms, inserting at the end of a string is obtained by passing Source'Last+1 as the Before parameter.

109 NOTE 3 If a null Character_Mapping_Function is passed to any of the string handling subprograms, Constraint_Error is propagated.

A.4.4 Bounded-Length String Handling

1/5 The language-defined package Strings.Bounded provides a generic package each of whose instances yields a private type Bounded_String and a set of operations. An object of a particular Bounded_String type represents a String whose low bound is 1 and whose length can vary conceptually between 0 and a maximum size established at the generic instantiation. The subprograms for fixed-length string handling are either overloaded directly for Bounded_String, or are modified as necessary to reflect the variability in

length. Additionally, since the Bounded_String type is private, appropriate constructor and selector operations are provided.

Static Semantics

The library package Strings.Bounded has the following declaration: 2

```
with Ada.Strings.Maps;                                                                    3/5
package Ada.Strings.Bounded
   with Preelaborate, Nonblocking, Global => in out synchronized is

   generic                                                                                4
      Max   : Positive;     -- Maximum length of a Bounded_String
   package Generic_Bounded_Length is

      Max_Length : constant Positive := Max;                                              5

      type Bounded_String is private;                                                     6

      Null_Bounded_String : constant Bounded_String;                                      7

      subtype Length_Range is Natural range 0 .. Max_Length;                              8

      function Length (Source : in Bounded_String) return Length_Range;                  9

   -- Conversion, Concatenation, and Selection functions                                  10

      function To_Bounded_String (Source : in String;                                     11
                                  Drop   : in Truncation := Error)
         return Bounded_String;

      function To_String (Source : in Bounded_String) return String;                     12

      procedure Set_Bounded_String                                                        12.1/2
         (Target :  out Bounded_String;
          Source : in    String;
          Drop   : in    Truncation := Error);

      function Append (Left, Right : in Bounded_String;                                   13
                       Drop        : in Truncation  := Error)
         return Bounded_String;

      function Append (Left  : in Bounded_String;                                         14
                       Right : in String;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      function Append (Left  : in String;                                                 15
                       Right : in Bounded_String;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      function Append (Left  : in Bounded_String;                                         16
                       Right : in Character;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      function Append (Left  : in Character;                                              17
                       Right : in Bounded_String;
                       Drop  : in Truncation := Error)
         return Bounded_String;

      procedure Append (Source   : in out Bounded_String;                                 18
                        New_Item : in Bounded_String;
                        Drop     : in Truncation  := Error);

      procedure Append (Source   : in out Bounded_String;                                 19
                        New_Item : in String;
                        Drop     : in Truncation  := Error);

      procedure Append (Source   : in out Bounded_String;                                 20
                        New_Item : in Character;
                        Drop     : in Truncation  := Error);

      function "&" (Left, Right : in Bounded_String)                                      21
         return Bounded_String;
```

```
22          function "&" (Left : in Bounded_String; Right : in String)
               return Bounded_String;
23          function "&" (Left : in String; Right : in Bounded_String)
               return Bounded_String;
24          function "&" (Left : in Bounded_String; Right : in Character)
               return Bounded_String;
25          function "&" (Left : in Character; Right : in Bounded_String)
               return Bounded_String;
26          function Element (Source : in Bounded_String;
                              Index  : in Positive)
               return Character;
27          procedure Replace_Element (Source : in out Bounded_String;
                                       Index  : in Positive;
                                       By     : in Character);
28          function Slice (Source : in Bounded_String;
                            Low    : in Positive;
                            High   : in Natural)
               return String;
28.1/2      function Bounded_Slice
               (Source : in Bounded_String;
                Low    : in Positive;
                High   : in Natural)
                  return Bounded_String;
28.2/2      procedure Bounded_Slice
               (Source : in     Bounded_String;
                Target :    out Bounded_String;
                Low    : in     Positive;
                High   : in     Natural);
29          function "="  (Left, Right : in Bounded_String) return Boolean;
            function "="  (Left : in Bounded_String; Right : in String)
              return Boolean;
30          function "="  (Left : in String; Right : in Bounded_String)
              return Boolean;
31          function "<"  (Left, Right : in Bounded_String) return Boolean;
32          function "<"  (Left : in Bounded_String; Right : in String)
              return Boolean;
33          function "<"  (Left : in String; Right : in Bounded_String)
              return Boolean;
34          function "<=" (Left, Right : in Bounded_String) return Boolean;
35          function "<="  (Left : in Bounded_String; Right : in String)
              return Boolean;
36          function "<="  (Left : in String; Right : in Bounded_String)
              return Boolean;
37          function ">"  (Left, Right : in Bounded_String) return Boolean;
38          function ">"  (Left : in Bounded_String; Right : in String)
              return Boolean;
39          function ">"  (Left : in String; Right : in Bounded_String)
              return Boolean;
40          function ">=" (Left, Right : in Bounded_String) return Boolean;
41          function ">="  (Left : in Bounded_String; Right : in String)
              return Boolean;
42          function ">="  (Left : in String; Right : in Bounded_String)
              return Boolean;
43/2        -- Search subprograms
```

```
function Index (Source  : in Bounded_String;                                    43.1/2
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping := Maps.Identity)
   return Natural;
function Index (Source  : in Bounded_String;                                    43.2/6
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in not null Maps.Character_Mapping_Function)  |
   return Natural;
function Index (Source  : in Bounded_String;                                    44
                Pattern : in String;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping
                           := Maps.Identity)
   return Natural;
function Index (Source  : in Bounded_String;                                    45/6
                Pattern : in String;
                Going   : in Direction := Forward;
                Mapping : in not null Maps.Character_Mapping_Function)  |
   return Natural;
function Index (Source  : in Bounded_String;                                    45.1/2
                Set     : in Maps.Character_Set;
                From    : in Positive;
                Test    : in Membership := Inside;
                Going   : in Direction := Forward)
   return Natural;
function Index (Source : in Bounded_String;                                     46
                Set    : in Maps.Character_Set;
                Test   : in Membership := Inside;
                Going  : in Direction  := Forward)
   return Natural;
function Index_Non_Blank (Source : in Bounded_String;                           46.1/2
                          From   : in Positive;
                          Going  : in Direction := Forward)
   return Natural;
function Index_Non_Blank (Source : in Bounded_String;                           47
                          Going  : in Direction := Forward)
   return Natural;
function Count (Source  : in Bounded_String;                                    48
                Pattern : in String;
                Mapping : in Maps.Character_Mapping
                             := Maps.Identity)
   return Natural;
function Count (Source  : in Bounded_String;                                    49/6
                Pattern : in String;
                Mapping : in not null Maps.Character_Mapping_Function)  |
   return Natural;
function Count (Source  : in Bounded_String;                                    50
                Set     : in Maps.Character_Set)
   return Natural;
procedure Find_Token (Source : in Bounded_String;                               50.1/3
                      Set    : in Maps.Character_Set;
                      From   : in Positive;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```

```
51          procedure Find_Token (Source : in Bounded_String;
                                  Set    : in Maps.Character_Set;
                                  Test   : in Membership;
                                  First  : out Positive;
                                  Last   : out Natural);
52       -- String translation subprograms
53          function Translate (Source  : in Bounded_String;
                                Mapping : in Maps.Character_Mapping)
             return Bounded_String;
54          procedure Translate (Source  : in out Bounded_String;
                                 Mapping : in Maps.Character_Mapping);
55/6        function Translate
              (Source  : in Bounded_String;
               Mapping : in not null Maps.Character_Mapping_Function)
               return Bounded_String;
56/6        procedure Translate
              (Source  : in out Bounded_String;
               Mapping : in not null Maps.Character_Mapping_Function);
57       -- String transformation subprograms
58          function Replace_Slice (Source   : in Bounded_String;
                                    Low      : in Positive;
                                    High     : in Natural;
                                    By       : in String;
                                    Drop     : in Truncation := Error)
             return Bounded_String;
59          procedure Replace_Slice (Source   : in out Bounded_String;
                                     Low      : in Positive;
                                     High     : in Natural;
                                     By       : in String;
                                     Drop     : in Truncation := Error);
60          function Insert (Source   : in Bounded_String;
                             Before   : in Positive;
                             New_Item : in String;
                             Drop     : in Truncation := Error)
             return Bounded_String;
61          procedure Insert (Source   : in out Bounded_String;
                              Before   : in Positive;
                              New_Item : in String;
                              Drop     : in Truncation := Error);
62          function Overwrite (Source    : in Bounded_String;
                                Position  : in Positive;
                                New_Item  : in String;
                                Drop      : in Truncation := Error)
             return Bounded_String;
63          procedure Overwrite (Source    : in out Bounded_String;
                                 Position  : in Positive;
                                 New_Item  : in String;
                                 Drop      : in Truncation := Error);
64          function Delete (Source  : in Bounded_String;
                             From    : in Positive;
                             Through : in Natural)
             return Bounded_String;
65          procedure Delete (Source  : in out Bounded_String;
                              From    : in Positive;
                              Through : in Natural);
66       --String selector subprograms
```

```
      function Trim (Source : in Bounded_String;                                    67
                     Side   : in Trim_End)
         return Bounded_String;
      procedure Trim (Source : in out Bounded_String;
                      Side   : in Trim_End);

      function Trim (Source : in Bounded_String;                                    68
                     Left   : in Maps.Character_Set;
                     Right  : in Maps.Character_Set)
         return Bounded_String;

      procedure Trim (Source : in out Bounded_String;                               69
                      Left   : in Maps.Character_Set;
                      Right  : in Maps.Character_Set);

      function Head (Source : in Bounded_String;                                    70
                     Count  : in Natural;
                     Pad    : in Character  := Space;
                     Drop   : in Truncation := Error)
         return Bounded_String;

      procedure Head (Source : in out Bounded_String;                               71
                      Count  : in Natural;
                      Pad    : in Character  := Space;
                      Drop   : in Truncation := Error);

      function Tail (Source : in Bounded_String;                                    72
                     Count  : in Natural;
                     Pad    : in Character  := Space;
                     Drop   : in Truncation := Error)
         return Bounded_String;

      procedure Tail (Source : in out Bounded_String;                               73
                      Count  : in Natural;
                      Pad    : in Character  := Space;
                      Drop   : in Truncation := Error);

   --String constructor subprograms                                                  74

      function "*" (Left  : in Natural;                                             75
                    Right : in Character)
         return Bounded_String;

      function "*" (Left  : in Natural;                                             76
                    Right : in String)
         return Bounded_String;

      function "*" (Left  : in Natural;                                             77
                    Right : in Bounded_String)
         return Bounded_String;

      function Replicate (Count : in Natural;                                       78
                          Item  : in Character;
                          Drop  : in Truncation := Error)
         return Bounded_String;

      function Replicate (Count : in Natural;                                       79
                          Item  : in String;
                          Drop  : in Truncation := Error)
         return Bounded_String;

      function Replicate (Count : in Natural;                                       80
                          Item  : in Bounded_String;
                          Drop  : in Truncation := Error)
         return Bounded_String;

   private                                                                          81
      ... -- not specified by the language
   end Generic_Bounded_Length;

end Ada.Strings.Bounded;                                                            82
```

Null_Bounded_String represents the null string. If an object of type Bounded_String is not otherwise initialized, it will be initialized to the same value as Null_Bounded_String. 83

84 `function Length (Source : in Bounded_String) return Length_Range;`

85 The Length function returns the length of the string represented by Source.

86
```
function To_Bounded_String (Source : in String;
                            Drop   : in Truncation := Error)
   return Bounded_String;
```

87/3 If Source'Length <= Max_Length, then this function returns a Bounded_String that represents Source. Otherwise, the effect depends on the value of Drop:

88 • If Drop=Left, then the result is a Bounded_String that represents the string comprising the rightmost Max_Length characters of Source.

89 • If Drop=Right, then the result is a Bounded_String that represents the string comprising the leftmost Max_Length characters of Source.

90 • If Drop=Error, then Strings.Length_Error is propagated.

91 `function To_String (Source : in Bounded_String) return String;`

92 To_String returns the String value with lower bound 1 represented by Source. If B is a Bounded_String, then B = To_Bounded_String(To_String(B)).

92.1/2
```
procedure Set_Bounded_String
   (Target :    out Bounded_String;
    Source : in     String;
    Drop   : in     Truncation := Error);
```

92.2/2 Equivalent to Target := To_Bounded_String (Source, Drop);

93 Each of the Append functions returns a Bounded_String obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying To_Bounded_String to the concatenation result string, with Drop as provided to the Append function.

94 Each of the procedures Append(Source, New_Item, Drop) has the same effect as the corresponding assignment Source := Append(Source, New_Item, Drop).

95 Each of the "&" functions has the same effect as the corresponding Append function, with Error as the Drop parameter.

96
```
function Element (Source : in Bounded_String;
                  Index  : in Positive)
   return Character;
```

97 Returns the character at position Index in the string represented by Source; propagates Index_Error if Index > Length(Source).

98
```
procedure Replace_Element (Source : in out Bounded_String;
                           Index  : in Positive;
                           By     : in Character);
```

99 Updates Source such that the character at position Index in the string represented by Source is By; propagates Index_Error if Index > Length(Source).

```
function Slice (Source : in Bounded_String;                                    100
                Low    : in Positive;
                High   : in Natural)
   return String;
```

Returns the slice at positions Low through High in the string represented by Source; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). The bounds of the returned string are Low and High. 101/1

```
function Bounded_Slice                                                         101.1/2
   (Source : in Bounded_String;
    Low    : in Positive;
    High   : in Natural)
       return Bounded_String;
```

Returns the slice at positions Low through High in the string represented by Source as a bounded string; propagates Index_Error if Low > Length(Source)+1 or High > Length(Source). 101.2/2

```
procedure Bounded_Slice                                                        101.3/2
   (Source : in     Bounded_String;
    Target :    out Bounded_String;
    Low    : in     Positive;
    High   : in     Natural);
```

Equivalent to Target := Bounded_Slice (Source, Low, High); 101.4/2

Each of the functions "=", "<", ">", "<=", and ">=" returns the same result as the corresponding String operation applied to the String values given or represented by the two parameters. 102

Each of the search subprograms (Index, Index_Non_Blank, Count, Find_Token) has the same effect as the corresponding subprogram in Strings.Fixed applied to the string represented by the Bounded_String parameter. 103

Each of the Translate subprograms, when applied to a Bounded_String, has an analogous effect to the corresponding subprogram in Strings.Fixed. For the Translate function, the translation is applied to the string represented by the Bounded_String parameter, and the result is converted (via To_Bounded_String) to a Bounded_String. For the Translate procedure, the string represented by the Bounded_String parameter after the translation is given by the Translate function for fixed-length strings applied to the string represented by the original value of the parameter. 104

Each of the transformation subprograms (Replace_Slice, Insert, Overwrite, Delete), selector subprograms (Trim, Head, Tail), and constructor functions ("*") has an effect based on its corresponding subprogram in Strings.Fixed, and Replicate is based on Fixed."*". In the case of a function, the corresponding fixed-length string subprogram is applied to the string represented by the Bounded_String parameter. To_Bounded_String is applied the result string, with Drop (or Error in the case of Generic_Bounded_Length."*") determining the effect when the string length exceeds Max_Length. In the case of a procedure, the corresponding function in Strings.Bounded.Generic_Bounded_Length is applied, with the result assigned into the Source parameter. 105/1

Implementation Advice

Bounded string objects should not be implemented by implicit pointers and dynamic allocation. 106

A.4.5 Unbounded-Length String Handling

The language-defined package Strings.Unbounded provides a private type Unbounded_String and a set of operations. An object of type Unbounded_String represents a String whose low bound is 1 and whose 1/5

length can vary conceptually between 0 and Natural'Last. The subprograms for fixed-length string handling are either overloaded directly for Unbounded_String, or are modified as necessary to reflect the flexibility in length. Since the Unbounded_String type is private, relevant constructor and selector operations are provided.

Static Semantics

2 The library package Strings.Unbounded has the following declaration:

3/5
```
with Ada.Strings.Maps;
package Ada.Strings.Unbounded
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/5
```
   type Unbounded_String is private
      with Preelaborable_Initialization;
```

5
```
   Null_Unbounded_String : constant Unbounded_String;
```

6
```
   function Length (Source : in Unbounded_String) return Natural;
```

7
```
   type String_Access is access all String;
   procedure Free (X : in out String_Access);
```

8
```
   -- Conversion, Concatenation, and Selection functions
```

9
```
   function To_Unbounded_String (Source : in String)
      return Unbounded_String;
```

10
```
   function To_Unbounded_String (Length : in Natural)
      return Unbounded_String;
```

11
```
   function To_String (Source : in Unbounded_String) return String;
```

11.1/2
```
   procedure Set_Unbounded_String
     (Target :    out Unbounded_String;
      Source : in     String);
```

12
```
   procedure Append (Source   : in out Unbounded_String;
                     New_Item : in Unbounded_String);
```

13
```
   procedure Append (Source   : in out Unbounded_String;
                     New_Item : in String);
```

14
```
   procedure Append (Source   : in out Unbounded_String;
                     New_Item : in Character);
```

15
```
   function "&" (Left, Right : in Unbounded_String)
      return Unbounded_String;
```

16
```
   function "&" (Left : in Unbounded_String; Right : in String)
      return Unbounded_String;
```

17
```
   function "&" (Left : in String; Right : in Unbounded_String)
      return Unbounded_String;
```

18
```
   function "&" (Left : in Unbounded_String; Right : in Character)
      return Unbounded_String;
```

19
```
   function "&" (Left : in Character; Right : in Unbounded_String)
      return Unbounded_String;
```

20
```
   function Element (Source : in Unbounded_String;
                     Index  : in Positive)
      return Character;
```

21
```
   procedure Replace_Element (Source : in out Unbounded_String;
                              Index  : in Positive;
                              By     : in Character);
```

22
```
   function Slice (Source : in Unbounded_String;
                   Low    : in Positive;
                   High   : in Natural)
      return String;
```

```
function Unbounded_Slice                                                    22.1/2
   (Source : in Unbounded_String;
    Low    : in Positive;
    High   : in Natural)
      return Unbounded_String;

procedure Unbounded_Slice                                                   22.2/2
   (Source : in     Unbounded_String;
    Target :    out Unbounded_String;
    Low    : in     Positive;
    High   : in     Natural);

function "="  (Left, Right : in Unbounded_String) return Boolean;           23

function "="  (Left : in Unbounded_String; Right : in String)               24
  return Boolean;

function "="  (Left : in String; Right : in Unbounded_String)               25
  return Boolean;

function "<"  (Left, Right : in Unbounded_String) return Boolean;           26

function "<"  (Left : in Unbounded_String; Right : in String)               27
  return Boolean;

function "<"  (Left : in String; Right : in Unbounded_String)               28
  return Boolean;

function "<=" (Left, Right : in Unbounded_String) return Boolean;           29

function "<="  (Left : in Unbounded_String; Right : in String)              30
  return Boolean;

function "<="  (Left : in String; Right : in Unbounded_String)              31
  return Boolean;

function ">"  (Left, Right : in Unbounded_String) return Boolean;           32

function ">"  (Left : in Unbounded_String; Right : in String)               33
  return Boolean;

function ">"  (Left : in String; Right : in Unbounded_String)               34
  return Boolean;

function ">=" (Left, Right : in Unbounded_String) return Boolean;           35

function ">="  (Left : in Unbounded_String; Right : in String)              36
  return Boolean;

function ">="  (Left : in String; Right : in Unbounded_String)              37
  return Boolean;

-- Search subprograms                                                       38

function Index (Source  : in Unbounded_String;                              38.1/2
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in Maps.Character_Mapping := Maps.Identity)
   return Natural;

function Index (Source  : in Unbounded_String;                              38.2/6
                Pattern : in String;
                From    : in Positive;
                Going   : in Direction := Forward;
                Mapping : in not null Maps.Character_Mapping_Function)
   return Natural;

function Index (Source   : in Unbounded_String;                             39
                Pattern  : in String;
                Going    : in Direction := Forward;
                Mapping  : in Maps.Character_Mapping
                              := Maps.Identity)
   return Natural;
```

```
40/6        function Index (Source  : in Unbounded_String;
                            Pattern : in String;
                            Going   : in Direction := Forward;
                            Mapping : in not null Maps.Character_Mapping_Function)
               return Natural;
40.1/2      function Index (Source : in Unbounded_String;
                            Set    : in Maps.Character_Set;
                            From   : in Positive;
                            Test   : in Membership := Inside;
                            Going  : in Direction := Forward)
               return Natural;
41          function Index (Source : in Unbounded_String;
                            Set    : in Maps.Character_Set;
                            Test   : in Membership := Inside;
                            Going  : in Direction  := Forward) return Natural;
41.1/2      function Index_Non_Blank (Source : in Unbounded_String;
                                      From   : in Positive;
                                      Going  : in Direction := Forward)
               return Natural;
42          function Index_Non_Blank (Source : in Unbounded_String;
                                      Going  : in Direction := Forward)
               return Natural;
43          function Count (Source  : in Unbounded_String;
                            Pattern : in String;
                            Mapping : in Maps.Character_Mapping
                                       := Maps.Identity)
               return Natural;
44/6        function Count (Source  : in Unbounded_String;
                            Pattern : in String;
                            Mapping : in not null Maps.Character_Mapping_Function)
               return Natural;
45          function Count (Source  : in Unbounded_String;
                            Set     : in Maps.Character_Set)
               return Natural;
45.1/3      procedure Find_Token (Source : in Unbounded_String;
                                  Set    : in Maps.Character_Set;
                                  From   : in Positive;
                                  Test   : in Membership;
                                  First  : out Positive;
                                  Last   : out Natural);
46          procedure Find_Token (Source : in Unbounded_String;
                                  Set    : in Maps.Character_Set;
                                  Test   : in Membership;
                                  First  : out Positive;
                                  Last   : out Natural);
47          -- String translation subprograms
48          function Translate (Source  : in Unbounded_String;
                                Mapping : in Maps.Character_Mapping)
               return Unbounded_String;
49          procedure Translate (Source  : in out Unbounded_String;
                                 Mapping : in Maps.Character_Mapping);
50/6        function Translate
              (Source  : in Unbounded_String;
               Mapping : in not null Maps.Character_Mapping_Function)
               return Unbounded_String;
51/6        procedure Translate
              (Source  : in out Unbounded_String;
               Mapping : in not null Maps.Character_Mapping_Function);
```

```
-- String transformation subprograms                                    52
   function Replace_Slice (Source   : in Unbounded_String;              53
                           Low      : in Positive;
                           High     : in Natural;
                           By       : in String)
      return Unbounded_String;
   procedure Replace_Slice (Source   : in out Unbounded_String;         54
                            Low      : in Positive;
                            High     : in Natural;
                            By       : in String);
   function Insert (Source   : in Unbounded_String;                     55
                    Before   : in Positive;
                    New_Item : in String)
      return Unbounded_String;
   procedure Insert (Source   : in out Unbounded_String;                56
                     Before   : in Positive;
                     New_Item : in String);
   function Overwrite (Source    : in Unbounded_String;                 57
                       Position  : in Positive;
                       New_Item  : in String)
      return Unbounded_String;
   procedure Overwrite (Source    : in out Unbounded_String;            58
                        Position  : in Positive;
                        New_Item  : in String);
   function Delete (Source  : in Unbounded_String;                      59
                    From    : in Positive;
                    Through : in Natural)
      return Unbounded_String;
   procedure Delete (Source  : in out Unbounded_String;                 60
                     From    : in Positive;
                     Through : in Natural);
   function Trim (Source : in Unbounded_String;                         61
                  Side   : in Trim_End)
      return Unbounded_String;
   procedure Trim (Source : in out Unbounded_String;                    62
                   Side   : in Trim_End);
   function Trim (Source : in Unbounded_String;                         63
                  Left   : in Maps.Character_Set;
                  Right  : in Maps.Character_Set)
      return Unbounded_String;
   procedure Trim (Source : in out Unbounded_String;                    64
                   Left   : in Maps.Character_Set;
                   Right  : in Maps.Character_Set);
   function Head (Source : in Unbounded_String;                         65
                  Count  : in Natural;
                  Pad    : in Character := Space)
      return Unbounded_String;
   procedure Head (Source : in out Unbounded_String;                    66
                   Count  : in Natural;
                   Pad    : in Character := Space);
   function Tail (Source : in Unbounded_String;                         67
                  Count  : in Natural;
                  Pad    : in Character := Space)
      return Unbounded_String;
   procedure Tail (Source : in out Unbounded_String;                    68
                   Count  : in Natural;
                   Pad    : in Character := Space);
```

```
69          function "*" (Left  : in Natural;
                          Right : in Character)
               return Unbounded_String;
70          function "*" (Left  : in Natural;
                          Right : in String)
               return Unbounded_String;
71          function "*" (Left  : in Natural;
                          Right : in Unbounded_String)
               return Unbounded_String;
72       private
            ... -- not specified by the language
         end Ada.Strings.Unbounded;
```

72.1/2 The type Unbounded_String needs finalization (see 7.6).

73 Null_Unbounded_String represents the null String. If an object of type Unbounded_String is not otherwise initialized, it will be initialized to the same value as Null_Unbounded_String.

74 The function Length returns the length of the String represented by Source.

75 The type String_Access provides a (nonprivate) access type for explicit processing of unbounded-length strings. The procedure Free performs an unchecked deallocation of an object of type String_Access.

76 The function To_Unbounded_String(Source : in String) returns an Unbounded_String that represents Source. The function To_Unbounded_String(Length : in Natural) returns an Unbounded_String that represents an uninitialized String whose length is Length.

77 The function To_String returns the String with lower bound 1 represented by Source. To_String and To_Unbounded_String are related as follows:

78 • If S is a String, then To_String(To_Unbounded_String(S)) = S.

79 • If U is an Unbounded_String, then To_Unbounded_String(To_String(U)) = U.

79.1/2 The procedure Set_Unbounded_String sets Target to an Unbounded_String that represents Source.

80 For each of the Append procedures, the resulting string represented by the Source parameter is given by the concatenation of the original value of Source and the value of New_Item.

81 Each of the "&" functions returns an Unbounded_String obtained by concatenating the string or character given or represented by one of the parameters, with the string or character given or represented by the other parameter, and applying To_Unbounded_String to the concatenation result string.

82 The Element, Replace_Element, and Slice subprograms have the same effect as the corresponding bounded-length string subprograms.

82.1/3 The function Unbounded_Slice returns the slice at positions Low through High in the string represented by Source as an Unbounded_String. The procedure Unbounded_Slice sets Target to the Unbounded_String representing the slice at positions Low through High in the string represented by Source. Both subprograms propagate Index_Error if Low > Length(Source)+1 or High > Length(Source).

83 Each of the functions "=", "<", ">", "<=", and ">=" returns the same result as the corresponding String operation applied to the String values given or represented by Left and Right.

84 Each of the search subprograms (Index, Index_Non_Blank, Count, Find_Token) has the same effect as the corresponding subprogram in Strings.Fixed applied to the string represented by the Unbounded_String parameter.

The Translate function has an analogous effect to the corresponding subprogram in Strings.Fixed. The translation is applied to the string represented by the Unbounded_String parameter, and the result is converted (via To_Unbounded_String) to an Unbounded_String. 85

Each of the transformation functions (Replace_Slice, Insert, Overwrite, Delete), selector functions (Trim, Head, Tail), and constructor functions ("*") is likewise analogous to its corresponding subprogram in Strings.Fixed. For each of the subprograms, the corresponding fixed-length string subprogram is applied to the string represented by the Unbounded_String parameter, and To_Unbounded_String is applied the result string. 86

For each of the procedures Translate, Replace_Slice, Insert, Overwrite, Delete, Trim, Head, and Tail, the resulting string represented by the Source parameter is given by the corresponding function for fixed-length strings applied to the string represented by Source's original value. 87

Implementation Requirements

No storage associated with an Unbounded_String object shall be lost upon assignment or scope exit. 88

A.4.6 String-Handling Sets and Mappings

The language-defined package Strings.Maps.Constants declares Character_Set and Character_Mapping constants corresponding to classification and conversion functions in package Characters.Handling. 1

Static Semantics

The library package Strings.Maps.Constants has the following declaration: 2

```
package Ada.Strings.Maps.Constants                                    3/5
   with Pure is

   Control_Set           : constant Character_Set;                    4
   Graphic_Set           : constant Character_Set;
   Letter_Set            : constant Character_Set;
   Lower_Set             : constant Character_Set;
   Upper_Set             : constant Character_Set;
   Basic_Set             : constant Character_Set;
   Decimal_Digit_Set     : constant Character_Set;
   Hexadecimal_Digit_Set : constant Character_Set;
   Alphanumeric_Set      : constant Character_Set;
   Special_Set           : constant Character_Set;
   ISO_646_Set           : constant Character_Set;

   Lower_Case_Map        : constant Character_Mapping;                5
     --Maps to lower case for letters, else identity
   Upper_Case_Map        : constant Character_Mapping;
     --Maps to upper case for letters, else identity
   Basic_Map             : constant Character_Mapping;
     --Maps to basic letter for letters, else identity

private                                                               6
   ... -- not specified by the language
end Ada.Strings.Maps.Constants;
```

Each of these constants represents a correspondingly named set of characters or character mapping in Characters.Handling (see A.3.2). 7

NOTE There are certain characters which are defined to be lower case letters by Unicode and are therefore allowed in identifiers, but are not considered lower case letters by Ada.Strings.Maps.Constants. 8/6

A.4.7 Wide_String Handling

1/3 Facilities for handling strings of Wide_Character elements are found in the packages Strings.Wide_Maps, Strings.Wide_Fixed, Strings.Wide_Bounded, Strings.Wide_Unbounded, and Strings.Wide_Maps.Wide_Constants, and in the library functions Strings.Wide_Hash, Strings.Wide_Fixed.Wide_Hash, Strings.Wide_Bounded.Wide_Hash, Strings.Wide_Unbounded.Wide_Hash, Strings.Wide_Hash_Case_Insensitive, Strings.Wide_Fixed.Wide_Hash_Case_Insensitive, Strings.Wide_Bounded.Wide_Hash_Case_Insensitive, Strings.Wide_Unbounded.Wide_Hash_Case_Insensitive, Strings.Wide_Equal_Case_Insensitive, Strings.Wide_Fixed.Wide_Equal_Case_Insensitive, Strings.Wide_Bounded.Wide_Equal_Case_Insensitive, and Strings.Wide_Unbounded.Wide_Equal_Case_Insensitive. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements.

Static Semantics

2 The package Strings.Wide_Maps has the following declaration.

3/5
```
package Ada.Strings.Wide_Maps
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/5
```
   -- Representation for a set of Wide_Character values:
   type Wide_Character_Set is private
      with Preelaborable_Initialization;
```

5
```
   Null_Set : constant Wide_Character_Set;
```

6
```
   type Wide_Character_Range is
     record
         Low  : Wide_Character;
         High : Wide_Character;
     end record;
   -- Represents Wide_Character range Low..High
```

7
```
   type Wide_Character_Ranges is array (Positive range <>)
      of Wide_Character_Range;
```

8
```
   function To_Set    (Ranges : in Wide_Character_Ranges)
      return Wide_Character_Set;
```

9
```
   function To_Set    (Span   : in Wide_Character_Range)
      return Wide_Character_Set;
```

10
```
   function To_Ranges (Set    : in Wide_Character_Set)
      return Wide_Character_Ranges;
```

11
```
   function "="   (Left, Right : in Wide_Character_Set) return Boolean;
```

12
```
   function "not" (Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "and" (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "or"  (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "xor" (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
   function "-"   (Left, Right : in Wide_Character_Set)
      return Wide_Character_Set;
```

13
```
   function Is_In (Element : in Wide_Character;
                   Set     : in Wide_Character_Set)
      return Boolean;
```

14
```
   function Is_Subset (Elements : in Wide_Character_Set;
                       Set      : in Wide_Character_Set)
      return Boolean;
```

15
```
   function "<=" (Left  : in Wide_Character_Set;
                  Right : in Wide_Character_Set)
      return Boolean renames Is_Subset;
```

```
   -- Alternative representation for a set of Wide_Character values:
   subtype Wide_Character_Sequence is Wide_String;

   function To_Set (Sequence  : in Wide_Character_Sequence)
      return Wide_Character_Set;

   function To_Set (Singleton : in Wide_Character)
      return Wide_Character_Set;

   function To_Sequence (Set  : in Wide_Character_Set)
      return Wide_Character_Sequence;

   -- Representation for a Wide_Character to Wide_Character mapping:
   type Wide_Character_Mapping is private
       with Preelaborable_Initialization;

   function Value (Map     : in Wide_Character_Mapping;
                   Element : in Wide_Character)
      return Wide_Character;

   Identity : constant Wide_Character_Mapping;

   function To_Mapping (From, To : in Wide_Character_Sequence)
      return Wide_Character_Mapping;

   function To_Domain (Map : in Wide_Character_Mapping)
      return Wide_Character_Sequence;

   function To_Range  (Map : in Wide_Character_Mapping)
      return Wide_Character_Sequence;

   type Wide_Character_Mapping_Function is
      access function (From : in Wide_Character) return Wide_Character;

private
   ... -- not specified by the language
end Ada.Strings.Wide_Maps;
```

The context clause for each of the packages Strings.Wide_Fixed, Strings.Wide_Bounded, and Strings.Wide_Unbounded identifies Strings.Wide_Maps instead of Strings.Maps.

Types Wide_Character_Set and Wide_Character_Mapping need finalization.

For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.Maps.Constants, and for library functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, Strings.Unbounded.Hash, Strings.Hash_Case_Insensitive, Strings.Fixed.Hash_Case_Insensitive, Strings.Bounded.Hash_Case_Insensitive, Strings.Unbounded.Hash_Case_Insensitive, Strings.Equal_Case_Insensitive, Strings.Fixed.Equal_Case_Insensitive, Strings.Bounded.Equal_Case_Insensitive, and Strings.Unbounded.Equal_Case_Insensitive, the corresponding wide string package or function has the same contents except that

- Wide_Space replaces Space
- Wide_Character replaces Character
- Wide_String replaces String
- Wide_Character_Set replaces Character_Set
- Wide_Character_Mapping replaces Character_Mapping
- Wide_Character_Mapping_Function replaces Character_Mapping_Function
- Wide_Maps replaces Maps
- Bounded_Wide_String replaces Bounded_String
- Null_Bounded_Wide_String replaces Null_Bounded_String
- To_Bounded_Wide_String replaces To_Bounded_String

40 • To_Wide_String replaces To_String

40.1/2 • Set_Bounded_Wide_String replaces Set_Bounded_String

41 • Unbounded_Wide_String replaces Unbounded_String

42 • Null_Unbounded_Wide_String replaces Null_Unbounded_String

43 • Wide_String_Access replaces String_Access

44 • To_Unbounded_Wide_String replaces To_Unbounded_String

44.1/2 • Set_Unbounded_Wide_String replaces Set_Unbounded_String

45 The following additional declaration is present in Strings.Wide_Maps.Wide_Constants:

46/2
```
Character_Set : constant Wide_Maps.Wide_Character_Set;
--Contains each Wide_Character value WC such that
--Characters.Conversions.Is_Character(WC) is True
```

46.1/2 Each Wide_Character_Set constant in the package Strings.Wide_Maps.Wide_Constants contains no values outside the Character portion of Wide_Character. Similarly, each Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Character.

47/5 Aspect Pure is replaced by aspects `Preelaborate, Nonblocking, Global => in out synchronized` in Strings.Wide_Maps.Wide_Constants.

48/5 NOTE If a null Wide_Character_Mapping_Function is passed to any of the Wide_String handling subprograms, Constraint_Error is propagated.

A.4.8 Wide_Wide_String Handling

1/3 Facilities for handling strings of Wide_Wide_Character elements are found in the packages Strings.Wide_Wide_Maps, Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, Strings.Wide_Wide_Unbounded, and Strings.Wide_Wide_Maps.Wide_Wide_Constants, and in the library functions Strings.Wide_Wide_Hash, Strings.Wide_Wide_Fixed.Wide_Wide_Hash, Strings.Wide_Wide_Bounded.Wide_Wide_Hash, Strings.Wide_Wide_Unbounded.Wide_Wide_Hash, Strings.Wide_Wide_Hash_Case_Insensitive, Strings.Wide_Wide_Fixed.Wide_Wide_Hash_Case_Insensitive, Strings.Wide_Wide_Bounded.Wide_Wide_Hash_Case_Insensitive, Strings.Wide_Wide_Unbounded.Wide_Wide_Hash_Case_Insensitive, Strings.Wide_Wide_Equal_Case_Insensitive, Strings.Wide_Wide_Fixed.Wide_Wide_Equal_Case_Insensitive, Strings.Wide_Wide_Bounded.Wide_Wide_Equal_Case_Insensitive, and Strings.Wide_Wide_Unbounded.Wide_Wide_Equal_Case_Insensitive. They provide the same string-handling operations as the corresponding packages and functions for strings of Character elements.

Static Semantics

2/2 The library package Strings.Wide_Wide_Maps has the following declaration.

3/5
```
package Ada.Strings.Wide_Wide_Maps
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/5
```
   -- Representation for a set of Wide_Wide_Character values:
   type Wide_Wide_Character_Set is private
      with Preelaborable_Initialization;
```

5/2
```
   Null_Set : constant Wide_Wide_Character_Set;
```

```
type Wide_Wide_Character_Range is                                                    6/2
   record
      Low  : Wide_Wide_Character;
      High : Wide_Wide_Character;
   end record;
-- Represents Wide_Wide_Character range Low..High
type Wide_Wide_Character_Ranges is array (Positive range <>)                         7/2
      of Wide_Wide_Character_Range;
function To_Set (Ranges : in Wide_Wide_Character_Ranges)                             8/2
      return Wide_Wide_Character_Set;
function To_Set (Span : in Wide_Wide_Character_Range)                                9/2
      return Wide_Wide_Character_Set;
function To_Ranges (Set : in Wide_Wide_Character_Set)                               10/2
      return Wide_Wide_Character_Ranges;
function "=" (Left, Right : in Wide_Wide_Character_Set) return Boolean;             11/2
function "not" (Right : in Wide_Wide_Character_Set)                                 12/2
      return Wide_Wide_Character_Set;
function "and" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function "or" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function "xor" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function "-" (Left, Right : in Wide_Wide_Character_Set)
      return Wide_Wide_Character_Set;
function Is_In (Element : in Wide_Wide_Character;                                   13/2
                Set     : in Wide_Wide_Character_Set)
      return Boolean;
function Is_Subset (Elements : in Wide_Wide_Character_Set;                          14/2
                    Set      : in Wide_Wide_Character_Set)
      return Boolean;
function "<=" (Left  : in Wide_Wide_Character_Set;                                  15/2
               Right : in Wide_Wide_Character_Set)
      return Boolean renames Is_Subset;
-- Alternative representation for a set of Wide_Wide_Character values:              16/2
subtype Wide_Wide_Character_Sequence is Wide_Wide_String;
function To_Set (Sequence : in Wide_Wide_Character_Sequence)                        17/2
      return Wide_Wide_Character_Set;
function To_Set (Singleton : in Wide_Wide_Character)                                18/2
      return Wide_Wide_Character_Set;
function To_Sequence (Set : in Wide_Wide_Character_Set)                             19/2
      return Wide_Wide_Character_Sequence;
-- Representation for a Wide_Wide_Character to Wide_Wide_Character                  20/5
-- mapping:
type Wide_Wide_Character_Mapping is private
   with Preelaborable_Initialization;
function Value (Map     : in Wide_Wide_Character_Mapping;                           21/2
                Element : in Wide_Wide_Character)
      return Wide_Wide_Character;
Identity : constant Wide_Wide_Character_Mapping;                                    22/2
function To_Mapping (From, To : in Wide_Wide_Character_Sequence)                    23/2
      return Wide_Wide_Character_Mapping;
function To_Domain (Map : in Wide_Wide_Character_Mapping)                           24/2
      return Wide_Wide_Character_Sequence;
function To_Range (Map : in Wide_Wide_Character_Mapping)                            25/2
      return Wide_Wide_Character_Sequence;
```

26/2
```
      type Wide_Wide_Character_Mapping_Function is
            access function (From : in Wide_Wide_Character)
            return Wide_Wide_Character;
```

27/2
```
   private
      ... -- not specified by the language
   end Ada.Strings.Wide_Wide_Maps;
```

28/2 The context clause for each of the packages Strings.Wide_Wide_Fixed, Strings.Wide_Wide_Bounded, and Strings.Wide_Wide_Unbounded identifies Strings.Wide_Wide_Maps instead of Strings.Maps.

28.1/3 Types Wide_Wide_Character_Set and Wide_Wide_Character_Mapping need finalization.

29/3 For each of the packages Strings.Fixed, Strings.Bounded, Strings.Unbounded, and Strings.-Maps.Constants, and for library functions Strings.Hash, Strings.Fixed.Hash, Strings.Bounded.Hash, Strings.Unbounded.Hash, Strings.Hash_Case_Insensitive, Strings.Fixed.Hash_Case_Insensitive, Strings.-Bounded.Hash_Case_Insensitive, Strings.Unbounded.Hash_Case_Insensitive, Strings.Equal_Case_-Insensitive, Strings.Fixed.Equal_Case_Insensitive, Strings.Bounded.Equal_Case_Insensitive, and Strings.-Unbounded.Equal_Case_Insensitive, the corresponding wide wide string package or function has the same contents except that

30/2 • Wide_Wide_Space replaces Space

31/2 • Wide_Wide_Character replaces Character

32/2 • Wide_Wide_String replaces String

33/2 • Wide_Wide_Character_Set replaces Character_Set

34/2 • Wide_Wide_Character_Mapping replaces Character_Mapping

35/2 • Wide_Wide_Character_Mapping_Function replaces Character_Mapping_Function

36/2 • Wide_Wide_Maps replaces Maps

37/2 • Bounded_Wide_Wide_String replaces Bounded_String

38/2 • Null_Bounded_Wide_Wide_String replaces Null_Bounded_String

39/2 • To_Bounded_Wide_Wide_String replaces To_Bounded_String

40/2 • To_Wide_Wide_String replaces To_String

41/2 • Set_Bounded_Wide_Wide_String replaces Set_Bounded_String

42/2 • Unbounded_Wide_Wide_String replaces Unbounded_String

43/2 • Null_Unbounded_Wide_Wide_String replaces Null_Unbounded_String

44/2 • Wide_Wide_String_Access replaces String_Access

45/2 • To_Unbounded_Wide_Wide_String replaces To_Unbounded_String

46/2 • Set_Unbounded_Wide_Wide_String replaces Set_Unbounded_String

47/2 The following additional declarations are present in Strings.Wide_Wide_Maps.Wide_Wide_Constants:

48/2
```
   Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
   -- Contains each Wide_Wide_Character value WWC such that
   -- Characters.Conversions.Is_Character(WWC) is True
   Wide_Character_Set : constant Wide_Wide_Maps.Wide_Wide_Character_Set;
   -- Contains each Wide_Wide_Character value WWC such that
   -- Characters.Conversions.Is_Wide_Character(WWC) is True
```

49/2 Each Wide_Wide_Character_Set constant in the package Strings.Wide_Wide_Maps.Wide_Wide_-Constants contains no values outside the Character portion of Wide_Wide_Character. Similarly, each

Wide_Wide_Character_Mapping constant in this package is the identity mapping when applied to any element outside the Character portion of Wide_Wide_Character.

Aspect Pure is replaced by aspects `Preelaborate, Nonblocking, Global => in out synchronized` in Strings.Wide_Wide_Maps.Wide_Wide_Constants. 50/5

NOTE If a null Wide_Wide_Character_Mapping_Function is passed to any of the Wide_Wide_String handling subprograms, Constraint_Error is propagated. 51/2

A.4.9 String Hashing

Static Semantics

The library function Strings.Hash has the following declaration: 1/2

```
with Ada.Containers;                                                    2/5
function Ada.Strings.Hash (Key : String) return Containers.Hash_Type
   with Pure;
```

Returns an implementation-defined value which is a function of the value of Key. If *A* and *B* are strings such that *A* equals *B*, Hash(*A*) equals Hash(*B*). 3/2

The library function Strings.Fixed.Hash has the following declaration: 4/2

```
with Ada.Containers, Ada.Strings.Hash;                                  5/3
function Ada.Strings.Fixed.Hash (Key : String) return Containers.Hash_Type
   renames Ada.Strings.Hash;
```

The generic library function Strings.Bounded.Hash has the following declaration: 6/2

```
with Ada.Containers;                                                    7/5
generic
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash (Key : Bounded.Bounded_String)
   return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

Equivalent to Strings.Hash (Bounded.To_String (Key)); 8/3

The library function Strings.Unbounded.Hash has the following declaration: 9/2

```
with Ada.Containers;                                                    10/5
function Ada.Strings.Unbounded.Hash (Key : Unbounded_String)
   return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

Equivalent to Strings.Hash (To_String (Key)); 11/3

The library function Strings.Hash_Case_Insensitive has the following declaration: 11.1/3

```
with Ada.Containers;                                                    11.2/5
function Ada.Strings.Hash_Case_Insensitive (Key : String)
   return Containers.Hash_Type
   with Pure;
```

Returns an implementation-defined value which is a function of the value of Key, converted to lower case. If A and B are strings such that Strings.Equal_Case_Insensitive (A, B) (see A.4.10) is True, then Hash_Case_Insensitive(A) equals Hash_Case_Insensitive(B). 11.3/3

The library function Strings.Fixed.Hash_Case_Insensitive has the following declaration: 11.4/3

```
with Ada.Containers, Ada.Strings.Hash_Case_Insensitive;                 11.5/3
function Ada.Strings.Fixed.Hash_Case_Insensitive (Key : String)
   return Containers.Hash_Type renames Ada.Strings.Hash_Case_Insensitive;
```

11.6/3 The generic library function Strings.Bounded.Hash_Case_Insensitive has the following declaration:

11.7/5
```
with Ada.Containers;
generic
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Hash_Case_Insensitive
   (Key : Bounded.Bounded_String) return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

11.8/3 Equivalent to Strings.Hash_Case_Insensitive (Bounded.To_String (Key));

11.9/3 The library function Strings.Unbounded.Hash_Case_Insensitive has the following declaration:

11.10/5
```
with Ada.Containers;
function Ada.Strings.Unbounded.Hash_Case_Insensitive
   (Key : Unbounded_String) return Containers.Hash_Type
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

11.11/3 Equivalent to Strings.Hash_Case_Insensitive (To_String (Key));

Implementation Advice

12/2 The Hash functions should be good hash functions, returning a wide spread of values for different string values. It should be unlikely for similar strings to return the same value.

A.4.10 String Comparison

Static Semantics

1/3 The library function Strings.Equal_Case_Insensitive has the following declaration:

2/5
```
function Ada.Strings.Equal_Case_Insensitive (Left, Right : String)
   return Boolean with Pure;
```

3/6 Returns True if the strings consist of the same sequence of characters after applying simple case folding, as defined by Section 3.13 of Unicode Version 15.0. Otherwise, returns False. This function uses the same method as is used to determine whether two identifiers are the same.

4/3 The library function Strings.Fixed.Equal_Case_Insensitive has the following declaration:

5/3
```
with Ada.Strings.Equal_Case_Insensitive;
function Ada.Strings.Fixed.Equal_Case_Insensitive
   (Left, Right : String) return Boolean
      renames Ada.Strings.Equal_Case_Insensitive;
```

6/3 The generic library function Strings.Bounded.Equal_Case_Insensitive has the following declaration:

7/5
```
generic
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Equal_Case_Insensitive
   (Left, Right : Bounded.Bounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

8/3 Equivalent to Strings.Equal_Case_Insensitive (Bounded.To_String (Left), Bounded.To_String (Right));

9/3 The library function Strings.Unbounded.Equal_Case_Insensitive has the following declaration:

10/5
```
function Ada.Strings.Unbounded.Equal_Case_Insensitive
   (Left, Right : Unbounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```

11/3 Equivalent to Strings.Equal_Case_Insensitive (To_String (Left), To_String (Right));

The library function Strings.Less_Case_Insensitive has the following declaration: 12/3

```
function Ada.Strings.Less_Case_Insensitive (Left, Right : String)
   return Boolean with Pure;
```
13/5

Performs a lexicographic comparison of strings Left and Right, converted to lower case. 14/3

The library function Strings.Fixed.Less_Case_Insensitive has the following declaration: 15/3

```
with Ada.Strings.Less_Case_Insensitive;
function Ada.Strings.Fixed.Less_Case_Insensitive
   (Left, Right : String) return Boolean
      renames Ada.Strings.Less_Case_Insensitive;
```
16/3

The generic library function Strings.Bounded.Less_Case_Insensitive has the following declaration: 17/3

```
generic
   with package Bounded is
      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
function Ada.Strings.Bounded.Less_Case_Insensitive
  (Left, Right : Bounded.Bounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```
18/5

Equivalent to Strings.Less_Case_Insensitive (Bounded.To_String (Left), Bounded.To_String (Right)); 19/3

The library function Strings.Unbounded.Less_Case_Insensitive has the following declaration: 20/3

```
function Ada.Strings.Unbounded.Less_Case_Insensitive
  (Left, Right : Unbounded_String) return Boolean
   with Preelaborate, Nonblocking, Global => in out synchronized;
```
21/5

Equivalent to Strings.Less_Case_Insensitive (To_String (Left), To_String (Right)); 22/3

A.4.11 String Encoding

1/3 Facilities for encoding, decoding, and converting strings in various character encoding schemes are provided by packages Strings.UTF_Encoding, Strings.UTF_Encoding.Conversions, Strings.-UTF_Encoding.Strings, Strings.UTF_Encoding.Wide_Strings, and Strings.UTF_Encoding.-Wide_Wide_Strings.

Static Semantics

2/3 The encoding library packages have the following declarations:

3/5
```
package Ada.Strings.UTF_Encoding
  with Pure is
```

4/3
```
   -- Declarations common to the string encoding packages
   type Encoding_Scheme is (UTF_8, UTF_16BE, UTF_16LE);
```

5/3
```
   subtype UTF_String is String;
```

6/3
```
   subtype UTF_8_String is String;
```

7/3
```
   subtype UTF_16_Wide_String is Wide_String;
```

8/3
```
   Encoding_Error : exception;
```

9/3
```
   BOM_8    : constant UTF_8_String :=
                 Character'Val(16#EF#) &
                 Character'Val(16#BB#) &
                 Character'Val(16#BF#);
```

10/3
```
   BOM_16BE : constant UTF_String :=
                 Character'Val(16#FE#) &
                 Character'Val(16#FF#);
```

11/3
```
   BOM_16LE : constant UTF_String :=
                 Character'Val(16#FF#) &
                 Character'Val(16#FE#);
```

12/3
```
   BOM_16   : constant UTF_16_Wide_String :=
                (1 => Wide_Character'Val(16#FEFF#));
```

13/3
```
   function Encoding (Item    : UTF_String;
                      Default : Encoding_Scheme := UTF_8)
      return Encoding_Scheme;
```

14/3
```
end Ada.Strings.UTF_Encoding;
```

15/5
```
package Ada.Strings.UTF_Encoding.Conversions
   with Pure is
```

16/3
```
   -- Conversions between various encoding schemes
   function Convert (Item          : UTF_String;
                     Input_Scheme  : Encoding_Scheme;
                     Output_Scheme : Encoding_Scheme;
                     Output_BOM    : Boolean := False) return UTF_String;
```

17/3
```
   function Convert (Item          : UTF_String;
                     Input_Scheme  : Encoding_Scheme;
                     Output_BOM    : Boolean := False)
      return UTF_16_Wide_String;
```

18/3
```
   function Convert (Item          : UTF_8_String;
                     Output_BOM    : Boolean := False)
      return UTF_16_Wide_String;
```

19/3
```
   function Convert (Item          : UTF_16_Wide_String;
                     Output_Scheme : Encoding_Scheme;
                     Output_BOM    : Boolean := False) return UTF_String;
```

20/3
```
   function Convert (Item          : UTF_16_Wide_String;
                     Output_BOM    : Boolean := False) return UTF_8_String;
```

21/3
```
end Ada.Strings.UTF_Encoding.Conversions;
```

```
package Ada.Strings.UTF_Encoding.Strings                                    22/5
   with Pure is

   -- Encoding / decoding between String and various encoding schemes      23/3
   function Encode (Item          : String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : String;                                   24/3
                    Output_BOM : Boolean  := False) return UTF_8_String;

   function Encode (Item       : String;                                   25/3
                    Output_BOM : Boolean  := False)
      return UTF_16_Wide_String;

   function Decode (Item         : UTF_String;                             26/3
                    Input_Scheme : Encoding_Scheme) return String;

   function Decode (Item : UTF_8_String) return String;                    27/3

   function Decode (Item : UTF_16_Wide_String) return String;              28/3

   function Decode (Item         : UTF_String;                             28.1/6
                    Input_Scheme : Encoding_Scheme;
                    Substitute   : Character) return String;

   function Decode (Item         : UTF_8_String;                           28.2/6
                    Substitute   : Character) return String;

   function Decode (Item         : UTF_16_Wide_String;                     28.3/6
                    Substitute   : Character) return String;

end Ada.Strings.UTF_Encoding.Strings;                                       29/3

package Ada.Strings.UTF_Encoding.Wide_Strings                               30/5
   with Pure is

   -- Encoding / decoding between Wide_String and various encoding schemes 31/3
   function Encode (Item          : Wide_String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : Wide_String;                              32/3
                    Output_BOM : Boolean  := False) return UTF_8_String;

   function Encode (Item       : Wide_String;                              33/3
                    Output_BOM : Boolean  := False)
      return UTF_16_Wide_String;

   function Decode (Item         : UTF_String;                             34/3
                    Input_Scheme : Encoding_Scheme) return Wide_String;

   function Decode (Item : UTF_8_String) return Wide_String;               35/3

   function Decode (Item : UTF_16_Wide_String) return Wide_String;         36/3

   function Decode (Item         : UTF_String;                             36.1/6
                    Input_Scheme : Encoding_Scheme;
                    Substitute   : Wide_Character) return Wide_String;

   function Decode (Item         : UTF_8_String;                           36.2/6
                    Substitute   : Wide_Character) return Wide_String;

   function Decode (Item         : UTF_16_Wide_String;                     36.3/6
                    Substitute   : Wide_Character) return Wide_String;

end Ada.Strings.UTF_Encoding.Wide_Strings;                                  37/3

package Ada.Strings.UTF_Encoding.Wide_Wide_Strings                          38/5
   with Pure is

   -- Encoding / decoding between Wide_Wide_String and various encoding schemes 39/3
   function Encode (Item          : Wide_Wide_String;
                    Output_Scheme : Encoding_Scheme;
                    Output_BOM    : Boolean  := False) return UTF_String;

   function Encode (Item       : Wide_Wide_String;                         40/3
                    Output_BOM : Boolean  := False) return UTF_8_String;
```

41/3
```
      function Encode (Item       : Wide_Wide_String;
                       Output_BOM : Boolean  := False)
         return UTF_16_Wide_String;
```

42/3
```
      function Decode (Item         : UTF_String;
                       Input_Scheme : Encoding_Scheme) return Wide_Wide_String;
```

43/3
```
      function Decode (Item : UTF_8_String) return Wide_Wide_String;
```

44/3
```
      function Decode (Item : UTF_16_Wide_String) return Wide_Wide_String;
```

45/3
```
   end Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
```

46/3 The type Encoding_Scheme defines encoding schemes. UTF_8 corresponds to the UTF-8 encoding scheme defined by Annex D of ISO/IEC 10646. UTF_16BE corresponds to the UTF-16 encoding scheme defined by Annex C of ISO/IEC 10646 in 8 bit, big-endian order; and UTF_16LE corresponds to the UTF-16 encoding scheme in 8 bit, little-endian order.

47/3 The subtype UTF_String is used to represent a String of 8-bit values containing a sequence of values encoded in one of three ways (UTF-8, UTF-16BE, or UTF-16LE). The subtype UTF_8_String is used to represent a String of 8-bit values containing a sequence of values encoded in UTF-8. The subtype UTF_16_Wide_String is used to represent a Wide_String of 16-bit values containing a sequence of values encoded in UTF-16.

48/3 The BOM_8, BOM_16BE, BOM_16LE, and BOM_16 constants correspond to values used at the start of a string to indicate the encoding.

49/3 Each of the Encode functions takes a String, Wide_String, or Wide_Wide_String Item parameter that is assumed to be an array of unencoded characters. Each of the Convert functions takes a UTF_String, UTF_8_String, or UTF_16_String Item parameter that is assumed to contain characters whose position values correspond to a valid encoding sequence according to the encoding scheme required by the function or specified by its Input_Scheme parameter.

50/3 Each of the Convert and Encode functions returns a UTF_String, UTF_8_String, or UTF_16_String value whose characters have position values that correspond to the encoding of the Item parameter according to the encoding scheme required by the function or specified by its Output_Scheme parameter. For UTF_8, no overlong encoding is returned. A BOM is included at the start of the returned string if the Output_BOM parameter is set to True. The lower bound of the returned string is 1.

51/3 Each of the Decode functions takes a UTF_String, UTF_8_String, or UTF_16_String Item parameter which is assumed to contain characters whose position values correspond to a valid encoding sequence according to the encoding scheme required by the function or specified by its Input_Scheme parameter, and returns the corresponding String, Wide_String, or Wide_Wide_String value. The lower bound of the returned string is 1.

52/3 For each of the Convert and Decode functions, an initial BOM in the input that matches the expected encoding scheme is ignored, and a different initial BOM causes Encoding_Error to be propagated.

53/3 The exception Encoding_Error is also propagated in the following situations:

54/4 - By a Convert or Decode function when a UTF encoded string contains an invalid encoding sequence.

55/4 - By a Convert or Decode function when the expected encoding is UTF-16BE or UTF-16LE and the input string has an odd length.

56/6 - By a Decode function without a Substitute parameter yielding a String when the decoding of a sequence results in a code point whose value exceeds 16#FF#.

- By a Decode function without a Substitute parameter yielding a Wide_String when the decoding of a sequence results in a code point whose value exceeds 16#FFFF#. 57/6

- By an Encode function taking a Wide_String as input when an invalid character appears in the input. In particular, the characters whose position is in the range 16#D800# .. 16#DFFF# are invalid because they conflict with UTF-16 surrogate encodings, and the characters whose position is 16#FFFE# or 16#FFFF# are also invalid because they conflict with BOM codes. 58/3

```
function Encoding (Item    : UTF_String;                                    59/3
                   Default : Encoding_Scheme := UTF_8)
   return Encoding_Scheme;
```

Inspects a UTF_String value to determine whether it starts with a BOM for UTF-8, UTF-16BE, or UTF_16LE. If so, returns the scheme corresponding to the BOM; otherwise, returns the value of Default. 60/3

```
function Convert (Item          : UTF_String;                              61/3
                  Input_Scheme  : Encoding_Scheme;
                  Output_Scheme : Encoding_Scheme;
                  Output_BOM    : Boolean := False) return UTF_String;
```

Returns the value of Item (originally encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme) encoded in one of these three schemes as specified by Output_Scheme. 62/3

```
function Convert (Item          : UTF_String;                              63/3
                  Input_Scheme  : Encoding_Scheme;
                  Output_BOM    : Boolean := False)
   return UTF_16_Wide_String;
```

Returns the value of Item (originally encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme) encoded in UTF-16. 64/3

```
function Convert (Item          : UTF_8_String;                            65/3
                  Output_BOM    : Boolean := False)
   return UTF_16_Wide_String;
```

Returns the value of Item (originally encoded in UTF-8) encoded in UTF-16. 66/3

```
function Convert (Item          : UTF_16_Wide_String;                      67/3
                  Output_Scheme : Encoding_Scheme;
                  Output_BOM    : Boolean := False) return UTF_String;
```

Returns the value of Item (originally encoded in UTF-16) encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme. 68/3

```
function Convert (Item          : UTF_16_Wide_String;                      69/3
                  Output_BOM    : Boolean := False) return UTF_8_String;
```

Returns the value of Item (originally encoded in UTF-16) encoded in UTF-8. 70/3

```
function Encode (Item          : String;                                  71/3
                 Output_Scheme : Encoding_Scheme;
                 Output_BOM    : Boolean  := False) return UTF_String;
```

Returns the value of Item encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme. 72/3

```
function Encode (Item       : String;                                      73/3
                 Output_BOM : Boolean  := False) return UTF_8_String;
```

Returns the value of Item encoded in UTF-8. 74/3

75/3
```
function Encode (Item       : String;
                 Output_BOM : Boolean  := False) return UTF_16_Wide_String;
```

76/3 Returns the value of Item encoded in UTF_16.

77/3
```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme) return String;
```

78/3 Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme.

79/3
```
function Decode (Item : UTF_8_String) return String;
```

80/3 Returns the result of decoding Item, which is encoded in UTF-8.

81/3
```
function Decode (Item : UTF_16_Wide_String) return String;
```

82/3 Returns the result of decoding Item, which is encoded in UTF-16.

82.1/6
```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme;
                 Substitute   : Character) return String;
```

82.2/6 Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme. If the decoding of a sequence results in a code point whose value exceeds 16#FF#, the resulting character is replaced by Substitute in the result String.

82.3/6
```
function Decode (Item         : UTF_8_String;
                 Substitute   : Character) return String;
```

82.4/6 Returns the result of decoding Item, which is encoded in UTF-8. If the decoding of a sequence results in a code point whose value exceeds 16#FF#, the resulting character is replaced by Substitute in the result String.

82.5/6
```
function Decode (Item         : UTF_16_Wide_String;
                 Substitute   : Character) return String;
```

82.6/6 Returns the result of decoding Item, which is encoded in UTF-16. If the decoding of a sequence results in a code point whose value exceeds 16#FF#, the resulting character is replaced by Substitute in the result String.

83/3
```
function Encode (Item          : Wide_String;
                 Output_Scheme : Encoding_Scheme;
                 Output_BOM    : Boolean  := False) return UTF_String;
```

84/3 Returns the value of Item encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme.

85/3
```
function Encode (Item       : Wide_String;
                 Output_BOM : Boolean  := False) return UTF_8_String;
```

86/3 Returns the value of Item encoded in UTF-8.

87/3
```
function Encode (Item       : Wide_String;
                 Output_BOM : Boolean  := False) return UTF_16_Wide_String;
```

88/3 Returns the value of Item encoded in UTF_16.

89/3
```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme) return Wide_String;
```

90/3 Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme.

```
function Decode (Item : UTF_8_String) return Wide_String;
```
91/3

Returns the result of decoding Item, which is encoded in UTF-8. 92/3

```
function Decode (Item : UTF_16_Wide_String) return Wide_String;
```
93/3

Returns the result of decoding Item, which is encoded in UTF-16. 94/3

```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme;
                 Substitute   : Wide_Character) return Wide_String;
```
94.1/6

Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme. If the decoding of a sequence results in a code point whose value exceeds 16#FFFF#, the resulting character is replaced by Substitute in the result Wide_String. 94.2/6

```
function Decode (Item         : UTF_8_String;
                 Substitute   : Wide_Character) return Wide_String;
```
94.3/6

Returns the result of decoding Item, which is encoded in UTF-8. If the decoding of a sequence results in a code point whose value exceeds 16#FFFF#, the resulting character is replaced by Substitute in the result Wide_String. 94.4/6

```
function Decode (Item         : UTF_16_Wide_String;
                 Substitute   : Wide_Character) return Wide_String;
```
94.5/6

Returns the result of decoding Item, which is encoded in UTF-16. If the decoding of a sequence results in a code point whose value exceeds 16#FFFF#, the resulting character is replaced by Substitute in the result Wide_String. 94.6/6

```
function Encode (Item          : Wide_Wide_String;
                 Output_Scheme : Encoding_Scheme;
                 Output_BOM    : Boolean  := False) return UTF_String;
```
95/3

Returns the value of Item encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Output_Scheme. 96/3

```
function Encode (Item       : Wide_Wide_String;
                 Output_BOM : Boolean  := False) return UTF_8_String;
```
97/3

Returns the value of Item encoded in UTF-8. 98/3

```
function Encode (Item       : Wide_Wide_String;
                 Output_BOM : Boolean  := False) return UTF_16_Wide_String;
```
99/3

Returns the value of Item encoded in UTF_16. 100/3

```
function Decode (Item         : UTF_String;
                 Input_Scheme : Encoding_Scheme) return Wide_Wide_String;
```
101/3

Returns the result of decoding Item, which is encoded in UTF-8, UTF-16LE, or UTF-16BE as specified by Input_Scheme. 102/3

```
function Decode (Item : UTF_8_String) return Wide_Wide_String;
```
103/3

Returns the result of decoding Item, which is encoded in UTF-8. 104/3

```
function Decode (Item : UTF_16_Wide_String) return Wide_Wide_String;
```
105/3

Returns the result of decoding Item, which is encoded in UTF-16. 106/3

Implementation Advice

107/3 If an implementation supports other encoding schemes, another similar child of Ada.Strings should be defined.

108/3 NOTE A BOM (Byte-Order Mark, code position 16#FEFF#) can be included in a file or other entity to indicate the encoding; it is skipped when decoding. Typically, only the first line of a file or other entity contains a BOM. When decoding, the Encoding function can be called on the first line to determine the encoding; this encoding will then be used in subsequent calls to Decode to convert all of the lines to an internal format.

A.4.12 Universal Text Buffers

1/5 A universal text buffer can be used to save and retrieve text of any language-defined string type. The types used to save and retrieve the text can be different.

Static Semantics

2/5 The text buffer library packages have the following declarations:

3/5
```
with Ada.Strings.UTF_Encoding.Wide_Wide_Strings;
package Ada.Strings.Text_Buffers
   with Pure is
```

4/5
```
   type Text_Buffer_Count is range 0 .. implementation-defined;
```

5/5
```
   New_Line_Count : constant Text_Buffer_Count := implementation-defined;
```

6/5
```
   type Root_Buffer_Type is abstract tagged private
      with Default_Initial_Condition =>
             Current_Indent (Root_Buffer_Type) = 0;
```

7/5
```
   procedure Put (
      Buffer : in out Root_Buffer_Type;
      Item   : in     String) is abstract;
```

8/5
```
   procedure Wide_Put (
      Buffer : in out Root_Buffer_Type;
      Item   : in     Wide_String) is abstract;
```

9/5
```
   procedure Wide_Wide_Put (
      Buffer : in out Root_Buffer_Type;
      Item   : in     Wide_Wide_String) is abstract;
```

10/5
```
   procedure Put_UTF_8 (
      Buffer : in out Root_Buffer_Type;
      Item   : in     UTF_Encoding.UTF_8_String) is abstract;
```

11/5
```
   procedure Wide_Put_UTF_16 (
      Buffer : in out Root_Buffer_Type;
      Item   : in     UTF_Encoding.UTF_16_Wide_String) is abstract;
```

12/5
```
   procedure New_Line (Buffer : in out Root_Buffer_Type) is abstract;
```

13/5
```
   Standard_Indent : constant Text_Buffer_Count := 3;
```

14/5
```
   function Current_Indent (
      Buffer : Root_Buffer_Type) return Text_Buffer_Count;
```

15/5
```
   procedure Increase_Indent (
      Buffer : in out Root_Buffer_Type;
      Amount : in     Text_Buffer_Count := Standard_Indent)
      with Post'Class =>
         Current_Indent (Buffer) = Current_Indent (Buffer)'Old + Amount;
```

```
   procedure Decrease_Indent (                                                    16/5
      Buffer : in out Root_Buffer_Type;
      Amount : in     Text_Buffer_Count := Standard_Indent)
      with Pre'Class =>
              Current_Indent (Buffer) >= Amount
                 or else raise Constraint_Error,
           Post'Class =>
              Current_Indent (Buffer) =
                 Current_Indent (Buffer)'Old - Amount;

private                                                                           17/5
   ... -- not specified by the language
end Ada.Strings.Text_Buffers;

package Ada.Strings.Text_Buffers.Unbounded                                        18/5
   with Preelaborate, Nonblocking, Global => null is

   type Buffer_Type is new Root_Buffer_Type with private;                         19/5

   function Get (                                                                 20/5
      Buffer : in out Buffer_Type)
      return String
      with Post'Class =>
         Get'Result'First = 1 and then Current_Indent (Buffer) = 0;

   function Wide_Get (                                                            21/5
      Buffer : in out Buffer_Type)
      return Wide_String
      with Post'Class =>
         Wide_Get'Result'First = 1 and then Current_Indent (Buffer) = 0;

   function Wide_Wide_Get (                                                       22/5
      Buffer : in out Buffer_Type)
      return Wide_Wide_String
      with Post'Class =>
         Wide_Wide_Get'Result'First = 1
            and then Current_Indent (Buffer) = 0;

   function Get_UTF_8 (                                                           23/5
      Buffer : in out Buffer_Type)
      return UTF_Encoding.UTF_8_String
      with Post'Class =>
         Get_UTF_8'Result'First = 1 and then Current_Indent (Buffer) = 0;

   function Wide_Get_UTF_16 (                                                     24/5
      Buffer : in out Buffer_Type)
      return UTF_Encoding.UTF_16_Wide_String
      with Post'Class =>
         Wide_Get_UTF_16'Result'First = 1
            and then Current_Indent (Buffer) = 0;

private                                                                           25/5
   ... -- not specified by the language, but will include nonabstract
       -- overridings of all inherited subprograms that require overriding.
end Ada.Strings.Text_Buffers.Unbounded;

package Ada.Strings.Text_Buffers.Bounded                                          26/5
   with Pure, Nonblocking, Global => null is

   type Buffer_Type (Max_Characters : Text_Buffer_Count)                          27/5
      is new Root_Buffer_Type with private
      with Default_Initial_Condition => not Text_Truncated (Buffer_Type);

   function Text_Truncated (Buffer : in Buffer_Type) return Boolean;              28/5

   -- Get, Wide_Get, Wide_Wide_Get, Get_UTF_8, and Wide_Get_UTF_16                29/5
   -- are declared here just as in the Unbounded child.

private                                                                           30/5
   ... -- not specified by the language, but will include nonabstract
       -- overridings of all inherited subprograms that require overriding.
end Ada.Strings.Text_Buffers.Bounded;
```

31/6 If an object of a text buffer type is not otherwise initialized, it is initialized to the empty state, with no stored characters.

32/5 New_Line stores New_Line_Count characters that represent a new line into a text buffer. Current_Indent returns the current indentation associated with the buffer, with zero meaning there is no indentation in effect; Increase_Indent and Decrease_Indent increase or decrease the indentation associated with the buffer.

33/5 A call to Put, Wide_Put, Wide_Wide_Put, Put_UTF_8, or Wide_Put_UTF_16 stores a sequence of characters into the text buffer, preceded by Current_Indent(Buffer) spaces (Wide_Wide_Characters with position 32) if there is at least one character in Item and it would have been the first character on the current line.

34/5 A call to function Get, Wide_Get, Wide_Wide_Get, Get_UTF_8, or Wide_Get_UTF_16 returns the same sequence of characters as was present in the calls that stored the characters into the buffer, if representable. For a call to Get, if any character in the sequence is not defined in Character, the result is implementation defined. Similarly, for a call to Wide_Get, if any character in the sequence is not defined in Wide_Character, the result is implementation defined. As part of a call on any of the Get functions, the buffer is reset to an empty state, with no stored characters.

35/5 In the case of a Buf of type Text_Buffers.Bounded.Buffer_Type, Text_Truncated (Buf) returns True if the various Put procedures together have attempted to store more than Buf.Max_Characters into Buf. If this function returns True, then the various Get functions return a representation of only the first Buf.Max_Characters characters that were stored in Buf.

Implementation Advice

36/5 Bounded buffer objects should be implemented without dynamic allocation.

A.5 The Numerics Packages

1 The library package Numerics is the parent of several child units that provide facilities for mathematical computation. One child, the generic package Generic_Elementary_Functions, is defined in A.5.1, together with nongeneric equivalents; two others, the package Float_Random and the generic package Discrete_Random, are defined in A.5.2. Additional (optional) children are defined in Annex G, “Numerics”.

Static Semantics

2/1 *This paragraph was deleted.*

3/5
```
package Ada.Numerics
   with Pure is
   Argument_Error : exception;
   Pi : constant :=
          3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
   π  : constant := Pi;
   e  : constant :=
          2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;
```

4 The Argument_Error exception is raised by a subprogram in a child unit of Numerics to signal that one or more of the actual subprogram parameters are outside the domain of the corresponding mathematical function.

Implementation Permissions

The implementation may specify the values of Pi and e to a larger number of significant digits. 5

A.5.1 Elementary Functions

Implementation-defined approximations to the mathematical functions known as the "elementary functions" are provided by the subprograms in Numerics.Generic_Elementary_Functions. Nongeneric equivalents of this generic package for each of the predefined floating point types are also provided as children of Numerics. 1

Static Semantics

The generic library package Numerics.Generic_Elementary_Functions has the following declaration: 2

```
generic                                                                    3/5
   type Float_Type is digits <>;

package Ada.Numerics.Generic_Elementary_Functions
   with Pure, Nonblocking is

   function Sqrt    (X           : Float_Type'Base) return Float_Type'Base;   4
   function Log     (X           : Float_Type'Base) return Float_Type'Base;
   function Log     (X, Base     : Float_Type'Base) return Float_Type'Base;
   function Exp     (X           : Float_Type'Base) return Float_Type'Base;
   function "**"    (Left, Right : Float_Type'Base) return Float_Type'Base;

   function Sin     (X           : Float_Type'Base) return Float_Type'Base;   5
   function Sin     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cos     (X           : Float_Type'Base) return Float_Type'Base;
   function Cos     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Tan     (X           : Float_Type'Base) return Float_Type'Base;
   function Tan     (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Cot     (X           : Float_Type'Base) return Float_Type'Base;
   function Cot     (X, Cycle    : Float_Type'Base) return Float_Type'Base;

   function Arcsin  (X           : Float_Type'Base) return Float_Type'Base;   6
   function Arcsin  (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X           : Float_Type'Base) return Float_Type'Base;
   function Arccos  (X, Cycle    : Float_Type'Base) return Float_Type'Base;
   function Arctan  (Y           : Float_Type'Base;
                     X           : Float_Type'Base := 1.0)
                                                    return Float_Type'Base;
   function Arctan  (Y           : Float_Type'Base;
                     X           : Float_Type'Base := 1.0;
                     Cycle       : Float_Type'Base) return Float_Type'Base;
   function Arccot  (X           : Float_Type'Base;
                     Y           : Float_Type'Base := 1.0)
                                                    return Float_Type'Base;
   function Arccot  (X           : Float_Type'Base;
                     Y           : Float_Type'Base := 1.0;
                     Cycle       : Float_Type'Base) return Float_Type'Base;

   function Sinh    (X           : Float_Type'Base) return Float_Type'Base;   7
   function Cosh    (X           : Float_Type'Base) return Float_Type'Base;
   function Tanh    (X           : Float_Type'Base) return Float_Type'Base;
   function Coth    (X           : Float_Type'Base) return Float_Type'Base;
   function Arcsinh (X           : Float_Type'Base) return Float_Type'Base;
   function Arccosh (X           : Float_Type'Base) return Float_Type'Base;
   function Arctanh (X           : Float_Type'Base) return Float_Type'Base;
   function Arccoth (X           : Float_Type'Base) return Float_Type'Base;

end Ada.Numerics.Generic_Elementary_Functions;                             8
```

The library package Numerics.Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Elementary_Functions, except that the predefined type Float is systematically substituted for Float_Type'Base throughout. Nongeneric equivalents of Numerics.Generic_Elementary_- 9/1

Functions for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Elementary_Functions, Numerics.Long_Elementary_Functions, etc.

10 The functions have their usual mathematical meanings. When the Base parameter is specified, the Log function computes the logarithm to the given base; otherwise, it computes the natural logarithm. When the Cycle parameter is specified, the parameter X of the forward trigonometric functions (Sin, Cos, Tan, and Cot) and the results of the inverse trigonometric functions (Arcsin, Arccos, Arctan, and Arccot) are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians.

11 The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch:

12 • The results of the Sqrt and Arccosh functions and that of the exponentiation operator are nonnegative.

13 • The result of the Arcsin function is in the quadrant containing the point (1.0, *x*), where *x* is the value of the parameter X. This quadrant is I or IV; thus, the range of the Arcsin function is approximately $-\pi/2.0$ to $\pi/2.0$ (–Cycle/4.0 to Cycle/4.0, if the parameter Cycle is specified).

14 • The result of the Arccos function is in the quadrant containing the point (*x*, 1.0), where *x* is the value of the parameter X. This quadrant is I or II; thus, the Arccos function ranges from 0.0 to approximately π (Cycle/2.0, if the parameter Cycle is specified).

15 • The results of the Arctan and Arccot functions are in the quadrant containing the point (*x*, *y*), where *x* and *y* are the values of the parameters X and Y, respectively. This may be any quadrant (I through IV) when the parameter X (resp., Y) of Arctan (resp., Arccot) is specified, but it is restricted to quadrants I and IV (resp., I and II) when that parameter is omitted. Thus, the range when that parameter is specified is approximately $-\pi$ to π (–Cycle/2.0 to Cycle/2.0, if the parameter Cycle is specified); when omitted, the range of Arctan (resp., Arccot) is that of Arcsin (resp., Arccos), as given above. When the point (*x*, *y*) lies on the negative x-axis, the result approximates

16 • π (resp., $-\pi$) when the sign of the parameter Y is positive (resp., negative), if Float_Type'Signed_Zeros is True;

17 • π, if Float_Type'Signed_Zeros is False.

18 (In the case of the inverse trigonometric functions, in which a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis.)

Dynamic Semantics

19 The exception Numerics.Argument_Error is raised, signaling a parameter value outside the domain of the corresponding mathematical function, in the following cases:

20 • by any forward or inverse trigonometric function with specified cycle, when the value of the parameter Cycle is zero or negative;

21 • by the Log function with specified base, when the value of the parameter Base is zero, one, or negative;

22 • by the Sqrt and Log functions, when the value of the parameter X is negative;

23 • by the exponentiation operator, when the value of the left operand is negative or when both operands have the value zero;

24 • by the Arcsin, Arccos, and Arctanh functions, when the absolute value of the parameter X exceeds one;

- by the Arctan and Arccot functions, when the parameters X and Y both have the value zero; 25
- by the Arccosh function, when the value of the parameter X is less than one; and 26
- by the Arccoth function, when the absolute value of the parameter X is less than one. 27

The exception Constraint_Error is raised, signaling a pole of the mathematical function (analogous to dividing by zero), in the following cases, provided that Float_Type'Machine_Overflows is True: 28

- by the Log, Cot, and Coth functions, when the value of the parameter X is zero; 29
- by the exponentiation operator, when the value of the left operand is zero and the value of the exponent is negative; 30
- by the Tan function with specified cycle, when the value of the parameter X is an odd multiple of the quarter cycle; 31
- by the Cot function with specified cycle, when the value of the parameter X is zero or a multiple of the half cycle; and 32
- by the Arctanh and Arccoth functions, when the absolute value of the parameter X is one. 33

Constraint_Error can also be raised when a finite result overflows (see G.2.4); this may occur for parameter values sufficiently *near* poles, and, in the case of some of the functions, for parameter values with sufficiently large magnitudes. When Float_Type'Machine_Overflows is False, the result at poles is unspecified. 34

When one parameter of a function with multiple parameters represents a pole and another is outside the function's domain, the latter takes precedence and Numerics.Argument_Error is raised. 35/5

Implementation Requirements

In the implementation of Numerics.Generic_Elementary_Functions, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Float_Type. 36

In the following cases, evaluation of an elementary function shall yield the *prescribed result*, provided that the preceding rules do not call for an exception to be raised: 37

- When the parameter X has the value zero, the Sqrt, Sin, Arcsin, Tan, Sinh, Arcsinh, Tanh, and Arctanh functions yield a result of zero, and the Exp, Cos, and Cosh functions yield a result of one. 38
- When the parameter X has the value one, the Sqrt function yields a result of one, and the Log, Arccos, and Arccosh functions yield a result of zero. 39
- When the parameter Y has the value zero and the parameter X has a positive value, the Arctan and Arccot functions yield a result of zero. 40
- The results of the Sin, Cos, Tan, and Cot functions with specified cycle are exact when the mathematical result is zero; those of the first two are also exact when the mathematical result is ± 1.0. 41
- Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero. 42

Other accuracy requirements for the elementary functions, which apply only in implementations conforming to the Numerics Annex, and then only in the “strict” mode defined there (see G.2), are given in G.2.4. 43

44 When Float_Type'Signed_Zeros is True, the sign of a zero result shall be as follows:

45 • A prescribed zero result delivered *at the origin* by one of the odd functions (Sin, Arcsin, Sinh, Arcsinh, Tan, Arctan or Arccot as a function of Y when X is fixed and positive, Tanh, and Arctanh) has the sign of the parameter X (Y, in the case of Arctan or Arccot).

46 • A prescribed zero result delivered by one of the odd functions *away from the origin*, or by some other elementary function, has an implementation-defined sign.

47/5 • A zero result that is not a prescribed result (that is, one that results from rounding or underflow) has the correct mathematical sign.

Implementation Permissions

48/5 The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

A.5.2 Random Number Generation

1 Facilities for the generation of pseudo-random floating point numbers are provided in the package Numerics.Float_Random; the generic package Numerics.Discrete_Random provides similar facilities for the generation of pseudo-random integers and pseudo-random values of enumeration types. For brevity, pseudo-random values of any of these types are called *random numbers*.

2 Some of the facilities provided are basic to all applications of random numbers. These include a limited private type each of whose objects serves as the generator of a (possibly distinct) sequence of random numbers; a function to obtain the "next" random number from a given sequence of random numbers (that is, from its generator); and subprograms to initialize or reinitialize a given generator to a time-dependent state or a state denoted by a single integer.

3 Other facilities are provided specifically for advanced applications. These include subprograms to save and restore the state of a given generator; a private type whose objects can be used to hold the saved state of a generator; and subprograms to obtain a string representation of a given generator state, or, given such a string representation, the corresponding state.

Static Semantics

4 The library package Numerics.Float_Random has the following declaration:

5/5
```
package Ada.Numerics.Float_Random
   with Global => in out synchronized is
```

6
```
   -- Basic facilities
```

7
```
   type Generator is limited private;
```

8/5
```
   subtype Uniformly_Distributed is Float range 0.0 .. 1.0;
   function Random (Gen : Generator) return Uniformly_Distributed
      with Global => overriding in out Gen;
```

9/5
```
   procedure Reset (Gen       : in Generator;
                    Initiator : in Integer)
      with Global => overriding in out Gen;
   procedure Reset (Gen       : in Generator)
      with Global => overriding in out Gen;
```

10
```
   -- Advanced facilities
```

11
```
   type State is private;
```

```
   procedure Save  (Gen        : in  Generator;                                  12/5
                    To_State   : out State);
   procedure Reset (Gen        : in  Generator;
                    From_State : in  State)
      with Global => overriding in out Gen;

   Max_Image_Width : constant := implementation-defined integer value;          13

   function Image (Of_State    : State)  return String;                         14
   function Value (Coded_State : String) return State;

private                                                                         15
   ... -- not specified by the language
end Ada.Numerics.Float_Random;
```

The type Generator needs finalization (see 7.6). 15.1/2

The generic library package Numerics.Discrete_Random has the following declaration: 16

```
generic                                                                         17/5
   type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random
   with Global => in out synchronized is

   -- Basic facilities                                                          18

   type Generator is limited private;                                           19

   function Random (Gen : Generator) return Result_Subtype                      20/5
      with Global => overriding in out Gen;

   function Random (Gen   : Generator;                                          20.1/5
                    First : Result_Subtype;
                    Last  : Result_Subtype) return Result_Subtype
      with Post => Random'Result in First .. Last,
           Global => overriding in out Gen;

   procedure Reset (Gen       : in Generator;                                   21/5
                    Initiator : in Integer)
      with Global => overriding in out Gen;
   procedure Reset (Gen       : in Generator)
      with Global => overriding in out Gen;

   -- Advanced facilities                                                       22

   type State is private;                                                       23

   procedure Save  (Gen        : in  Generator;                                 24/5
                    To_State   : out State);
   procedure Reset (Gen        : in  Generator;
                    From_State : in  State)
      with Global => overriding in out Gen;

   Max_Image_Width : constant := implementation-defined integer value;          25

   function Image (Of_State    : State)  return String;                         26
   function Value (Coded_State : String) return State;

private                                                                         27
   ... -- not specified by the language
end Ada.Numerics.Discrete_Random;
```

The type Generator needs finalization (see 7.6) in every instantiation of Numerics.Discrete_Random. 27.1/2

An object of the limited private type Generator is associated with a sequence of random numbers. Each generator has a hidden (internal) state, which the operations on generators use to determine the position in the associated sequence. All generators are implicitly initialized to an unspecified state that does not vary from one program execution to another; they may also be explicitly initialized, or reinitialized, to a time-dependent state, to a previously saved state, or to a state uniquely denoted by an integer value. 28

An object of the private type State can be used to hold the internal state of a generator. Such objects are only necessary if the application is designed to save and restore generator states or to examine or 29/5

manufacture them. The implicit initial value of type State corresponds to the implicit initial value of all generators.

30 The operations on generators affect the state and therefore the future values of the associated sequence. The semantics of the operations on generators and states are defined below.

31
```
function Random (Gen : Generator) return Uniformly_Distributed;
function Random (Gen : Generator) return Result_Subtype;
```

32/5 Obtains the "next" random number from the given generator, relative to its current state, according to an implementation-defined algorithm.

32.1/5
```
function Random (Gen   : Generator;
                 First : Result_Subtype;
                 Last  : Result_Subtype) return Result_Subtype
   with Post => Random'Result in First .. Last;
```

32.2/5 Obtains the "next" random number from the given generator, relative to its current state, according to an implementation-defined algorithm. If the range First .. Last is a null range, Constraint_Error is raised.

33
```
procedure Reset (Gen       : in Generator;
                 Initiator : in Integer);
procedure Reset (Gen       : in Generator);
```

34 Sets the state of the specified generator to one that is an unspecified function of the value of the parameter Initiator (or to a time-dependent state, if only a generator parameter is specified). The latter form of the procedure is known as the *time-dependent Reset procedure*.

35
```
procedure Save  (Gen        : in  Generator;
                 To_State   : out State);
procedure Reset (Gen        : in  Generator;
                 From_State : in  State);
```

36 Save obtains the current state of a generator. Reset gives a generator the specified state. A generator that is reset to a state previously obtained by invoking Save is restored to the state it had when Save was invoked.

37
```
function Image (Of_State    : State)  return String;
function Value (Coded_State : String) return State;
```

38 Image provides a representation of a state coded (in an implementation-defined way) as a string whose length is bounded by the value of Max_Image_Width. Value is the inverse of Image: Value(Image(S)) = S for each state S that can be obtained from a generator by invoking Save.

Dynamic Semantics

39 Instantiation of Numerics.Discrete_Random with a subtype having a null range raises Constraint_Error.

40/1 *This paragraph was deleted.*

Bounded (Run-Time) Errors

40.1/5 It is a bounded error to invoke Value with a string that is not the image of any generator state. If the error is detected, Constraint_Error or Program_Error is raised. Otherwise, a call to Reset with the resulting state will produce a generator such that calls to Random with this generator will produce a sequence of values of the appropriate subtype, but which are not necessarily random in character. That is, the sequence of values do not necessarily fulfill the implementation requirements of this subclause.

Implementation Requirements

Each call of a Random function has a *result range*; this is the range First .. Last for the version of Random with First and Last parameters and the range of the result subtype of the function otherwise. 40.2/5

A sufficiently long sequence of random numbers obtained by consecutive calls to Random that have the same generator and result range is approximately uniformly distributed over the result range. 41/5

A Random function in an instantiation of Numerics.Discrete_Random is guaranteed to yield each value in its result range in a finite number of calls, provided that the number of such values does not exceed 2^{15}. 42/5

Other performance requirements for the random number generator, which apply only in implementations conforming to the Numerics Annex, and then only in the "strict" mode defined there (see G.2), are given in G.2.5. 43

Documentation Requirements

No one algorithm for random number generation is best for all applications. To enable the user to determine the suitability of the random number generators for the intended application, the implementation shall describe the algorithm used and shall give its period, if known exactly, or a lower bound on the period, if the exact period is unknown. Periods that are so long that the periodicity is unobservable in practice can be described in such terms, without giving a numerical bound. 44

The implementation also shall document the minimum time interval between calls to the time-dependent Reset procedure that are guaranteed to initiate different sequences, and it shall document the nature of the strings that Value will accept without raising Constraint_Error. 45

Implementation Advice

Any storage associated with an object of type Generator should be reclaimed on exit from the scope of the object. 46

If the generator period is sufficiently long in relation to the number of distinct initiator values, then each possible value of Initiator passed to Reset should initiate a sequence of random numbers that does not, in a practical sense, overlap the sequence initiated by any other value. If this is not possible, then the mapping between initiator values and generator states should be a rapidly varying function of the initiator value. 47

NOTE 1 If two or more tasks are to share the same generator, then the tasks have to synchronize their access to the generator as for any shared variable (see 9.10). 48

NOTE 2 Within a given implementation, a repeatable random number sequence can be obtained by relying on the implicit initialization of generators or by explicitly initializing a generator with a repeatable initiator value. Different sequences of random numbers can be obtained from a given generator in different program executions by explicitly initializing the generator to a time-dependent state. 49

NOTE 3 A given implementation of the Random function in Numerics.Float_Random is not guaranteed to be capable of delivering the values 0.0 or 1.0. Applications will be more portable if they assume that these values, or values sufficiently close to them to behave indistinguishably from them, can occur. If a sequence of random integers from some range is necessary, the application should use one of the Random functions in an appropriate instantiation of Numerics.Discrete_Random, rather than transforming the result of the Random function in Numerics.Float_Random. 50/5

NOTE 4 Exponentially distributed (floating point) random numbers with mean and standard deviation 1.0 can be obtained by the transformation 52/5

```
-Log(Random(G) + Float'Model_Small)
```
53/2

where Log comes from Numerics.Elementary_Functions (see A.5.1); in this expression, the addition of Float'Model_Small avoids the exception that would be raised were Log to be given the value zero, without affecting the result (in most implementations) when Random returns a nonzero value. 54

Examples

55 *Example of a program that plays a simulated dice game:*

56
```
with Ada.Numerics.Discrete_Random;
procedure Dice_Game is
   subtype Die is Integer range 1 .. 6;
   subtype Dice is Integer range 2*Die'First .. 2*Die'Last;
   package Random_Die is new Ada.Numerics.Discrete_Random (Die);
   use Random_Die;
   G : Generator;
   D : Dice;
begin
   Reset (G);  -- Start the generator in a unique state in each run
   loop
      -- Roll a pair of dice; sum and process the results
      D := Random(G) + Random(G);
      ...
   end loop;
end Dice_Game;
```

57 *Example of a program that simulates coin tosses:*

58
```
with Ada.Numerics.Discrete_Random;
procedure Flip_A_Coin is
   type Coin is (Heads, Tails);
   package Random_Coin is new Ada.Numerics.Discrete_Random (Coin);
   use Random_Coin;
   G : Generator;
begin
   Reset (G);  -- Start the generator in a unique state in each run
   loop
      -- Toss a coin and process the result
      case Random(G) is
          when Heads =>
             ...
          when Tails =>
             ...
      end case;
   ...
   end loop;
end Flip_A_Coin;
```

Example of a parallel simulation of a physical system, with a separate generator of event probabilities in each task: 59

```
with Ada.Numerics.Float_Random;                                                    60
procedure Parallel_Simulation is
   use Ada.Numerics.Float_Random;
   task type Worker is
      entry Initialize_Generator (Initiator : in Integer);
      ...
   end Worker;
   W : array (1 .. 10) of Worker;
   task body Worker is
      G : Generator;
      Probability_Of_Event : Uniformly_Distributed;
   begin
      accept Initialize_Generator (Initiator : in Integer) do
         Reset (G, Initiator);
      end Initialize_Generator;
      loop
         ...
         Probability_Of_Event := Random(G);
         ...
      end loop;
   end Worker;
begin
   -- Initialize the generators in the Worker tasks to different states
   for I in W'Range loop
      W(I).Initialize_Generator (I);
   end loop;
   ... -- Wait for the Worker tasks to terminate
end Parallel_Simulation;
```

Although each Worker task initializes its generator to a different state, those states will be the same in every execution of the program. The generator states can be initialized uniquely in each program execution by instantiating Ada.Numerics.Discrete_Random for the type Integer in the main procedure, resetting the generator obtained from that instance to a time-dependent state, and then using random integers obtained from that generator to initialize the generators in each Worker task. 61/5

A.5.3 Attributes of Floating Point Types

Static Semantics

The following *representation-oriented attributes* are defined for every subtype S of a floating point type *T*. 1

S'Machine_Radix 2

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*.

The values of other representation-oriented attributes of a floating point subtype, and of the "primitive function" attributes of a floating point subtype described later, are defined in terms of a particular representation of nonzero values called the *canonical form*. The canonical form (for the type *T*) is the form 3

$$\pm\ mantissa \cdot T\text{'Machine\_Radix}^{exponent}$$

where

- *mantissa* is a fraction in the number base *T*'Machine_Radix, the first digit of which is nonzero, and 4
- *exponent* is an integer. 5

6 S'Machine_Mantissa
Yields the largest value of *p* such that every value expressible in the canonical form (for the type *T*), having a *p*-digit *mantissa* and an *exponent* between *T*'Machine_Emin and *T*'Machine_Emax, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

7 S'Machine_Emin
Yields the smallest (most negative) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

8 S'Machine_Emax
Yields the largest (most positive) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*.

9 S'Denorm Yields the value True if every value expressible in the form
$\pm\ mantissa \cdot T\text{'Machine\_Radix}^{T\text{'Machine\_Emin}}$
where *mantissa* is a nonzero *T*'Machine_Mantissa-digit fraction in the number base *T*'Machine_Radix, the first digit of which is zero, is a machine number (see 3.5.7) of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

10 The values described by the formula in the definition of S'Denorm are called *denormalized numbers*. A nonzero machine number that is not a denormalized number is a *normalized number*. A normalized number *x* of a given type *T* is said to be *represented in canonical form* when it is expressed in the canonical form (for the type *T*) with a *mantissa* having *T*'Machine_Mantissa digits; the resulting form is the *canonical-form representation* of *x*.

11 S'Machine_Rounds
Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

12 S'Machine_Overflows
Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

13 S'Signed_Zeros
Yields the value True if the hardware representation for the type *T* has the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type *T* as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

14 For every value *x* of a floating point type *T*, the *normalized exponent* of *x* is defined as follows:

15 • the normalized exponent of zero is (by convention) zero;

16 • for nonzero *x*, the normalized exponent of *x* is the unique integer *k* such that $T\text{'Machine\_Radix}^{k-1} \le |x| < T\text{'Machine\_Radix}^{k}$.

17 The following *primitive function attributes* are defined for any subtype S of a floating point type *T*.

18 S'Exponent S'Exponent denotes a function with the following specification:

```
function S'Exponent (X : T)
  return universal_integer
```

The function yields the normalized exponent of *X*.

S'Fraction

S'Fraction denotes a function with the following specification:

```
function S'Fraction (X : T)
  return T
```

The function yields the value $X \cdot T\text{'Machine\_Radix}^{-k}$, where *k* is the normalized exponent of *X*. A zero result, which can only occur when *X* is zero, has the sign of *X*.

S'Compose

S'Compose denotes a function with the following specification:

```
function S'Compose (Fraction : T;
                    Exponent : universal_integer)
  return T
```

Let *v* be the value $Fraction \cdot T\text{'Machine\_Radix}^{Exponent-k}$, where *k* is the normalized exponent of *Fraction*. If *v* is a machine number of the type *T*, or if $|v| \geq T\text{'Model\_Small}$, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *Fraction* when S'Signed_Zeros is True.

S'Scaling

S'Scaling denotes a function with the following specification:

```
function S'Scaling (X : T;
                    Adjustment : universal_integer)
  return T
```

Let *v* be the value $X \cdot T\text{'Machine\_Radix}^{Adjustment}$. If *v* is a machine number of the type *T*, or if $|v| \geq T\text{'Model\_Small}$, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True.

S'Floor

S'Floor denotes a function with the following specification:

```
function S'Floor (X : T)
  return T
```

The function yields the value $\lfloor X \rfloor$, that is, the largest (most positive) integral value less than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a positive sign.

S'Ceiling

S'Ceiling denotes a function with the following specification:

```
function S'Ceiling (X : T)
  return T
```

The function yields the value $\lceil X \rceil$, that is, the smallest (most negative) integral value greater than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a negative sign when S'Signed_Zeros is True.

S'Rounding

S'Rounding denotes a function with the following specification:

```
function S'Rounding (X : T)
  return T
```

The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

S'Unbiased_Rounding

S'Unbiased_Rounding denotes a function with the following specification:

```
function S'Unbiased_Rounding (X : T)
  return T
```

41 The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True.

41.1/2 S'Machine_Rounding

S'Machine_Rounding denotes a function with the following specification:

41.2/2
```
function S'Machine_Rounding (X : T)
  return T
```

41.3/2 The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor.

42 S'Truncation S'Truncation denotes a function with the following specification:

43
```
function S'Truncation (X : T)
  return T
```

44 The function yields the value $\lceil X \rceil$ when *X* is negative, and $\lfloor X \rfloor$ otherwise. A zero result has the sign of *X* when S'Signed_Zeros is True.

45 S'Remainder S'Remainder denotes a function with the following specification:

46
```
function S'Remainder (X, Y : T)
  return T
```

47 For nonzero *Y*, let *v* be the value $X - n \cdot Y$, where *n* is the integer nearest to the exact value of *X*/*Y*; if $|n - X/Y| = 1/2$, then *n* is chosen to be even. If *v* is a machine number of the type *T*, the function yields *v*; otherwise, it yields zero. Constraint_Error is raised if *Y* is zero. A zero result has the sign of *X* when S'Signed_Zeros is True.

48 S'Adjacent S'Adjacent denotes a function with the following specification:

49
```
function S'Adjacent (X, Towards : T)
  return T
```

50 If *Towards* = *X*, the function yields *X*; otherwise, it yields the machine number of the type *T* adjacent to *X* in the direction of *Towards*, if that machine number exists. If the result would be outside the base range of S, Constraint_Error is raised. When *T*'Signed_Zeros is True, a zero result has the sign of *X*. When *Towards* is zero, its sign has no bearing on the result.

51 S'Copy_Sign S'Copy_Sign denotes a function with the following specification:

52
```
function S'Copy_Sign (Value, Sign : T)
  return T
```

53 If the value of *Value* is nonzero, the function yields a result whose magnitude is that of *Value* and whose sign is that of *Sign*; otherwise, it yields the value zero. Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of *Sign* when S'Signed_Zeros is True.

54 S'Leading_Part

S'Leading_Part denotes a function with the following specification:

55
```
function S'Leading_Part (X : T;
                         Radix_Digits : universal_integer)
  return T
```

56 Let *v* be the value *T*'Machine_Radix$^{k-Radix_Digits}$, where *k* is the normalized exponent of *X*. The function yields the value

57 • $\lfloor X/v \rfloor \cdot v$, when *X* is nonnegative and *Radix_Digits* is positive;

58 • $\lceil X/v \rceil \cdot v$, when *X* is negative and *Radix_Digits* is positive.

Constraint_Error is raised when *Radix_Digits* is zero or negative. A zero result, which can only occur when *X* is zero, has the sign of *X*. 59

S'Machine S'Machine denotes a function with the following specification: 60

```
function S'Machine (X : T)
  return T
```
61

If *X* is a machine number of the type *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent machine numbers of the type *T*. Constraint_Error is raised if rounding or truncating *X* to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. 62

The following *model-oriented attributes* are defined for any subtype S of a floating point type *T*. 63

S'Model_Mantissa 64

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + 1$, where *d* is the requested decimal precision of *T*, and less than or equal to the value of *T*'Machine_Mantissa. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*.

S'Model_Emin 65

If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of *T*'Machine_Emin. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*.

S'Model_Epsilon 66

Yields the value $T\text{'Machine\_Radix}^{1 - T\text{'Model\_Mantissa}}$. The value of this attribute is of the type *universal_real*.

S'Model_Small 67

Yields the value $T\text{'Machine\_Radix}^{T\text{'Model\_Emin} - 1}$. The value of this attribute is of the type *universal_real*.

S'Model S'Model denotes a function with the following specification: 68

```
function S'Model (X : T)
  return T
```
69

If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. 70

S'Safe_First 71

Yields the lower bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*.

S'Safe_Last Yields the upper bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*. 72

A.5.4 Attributes of Fixed Point Types

Static Semantics

1 The following *representation-oriented* attributes are defined for every subtype S of a fixed point type *T*.

2 S'Machine_Radix
Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*.

3 S'Machine_Rounds
Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

4 S'Machine_Overflows
Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean.

5/6 A fixed point type *has integral values* if its Small value is an integer or the reciprocal of an integer. A generic formal type is defined to have integral values.

6/6 The following primitive function attributes are defined for every subtype S of a fixed point type *T* that has integral values.

7/6 S'Ceiling
S'Ceiling denotes a function with the following specification:

7.1/6
```
function S'Ceiling (X : T)
   return T
```

7.2/6 The function yields the value *Ceiling(X)*, that is, the smallest (most negative) integral value greater than or equal to *X*.

8/6 S'Floor
S'Floor denotes a function with the following specification:

8.1/6
```
function S'Floor (X : T)
   return T
```

8.2/6 The function yields the value *Floor(X)*, that is, the largest (most positive) integral value less than or equal to *X*.

9/6 S'Rounding
S'Rounding denotes a function with the following specification:

9.1/6
```
function S'Rounding (X : T)
   return T
```

9.2/6 The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers.

10/6 S'Unbiased_Rounding
S'Unbiased_Rounding denotes a function with the following specification:

10.1/6
```
function S'Unbiased_Rounding (X : T)
   return T
```

10.2/6 The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers.

11/6 S'Machine_Rounding
S'Machine_Rounding denotes a function with the following specification:

```
function S'Machine_Rounding (X : T)
   return T
```

11.1/6

The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. This function provides access to the rounding behavior which is most efficient on the target processor.

11.2/6

S'Truncation

12/6

S'Truncation denotes a function with the following specification:

```
function S'Truncation (X : T)
   return T
```

12.1/6

The function yields the value *Ceiling(X)* when *X* is negative, and *Floor(X)* otherwise.

12.2/6

For all of these attributes, if *T* is a generic formal type, Program_Error is raised if the actual subtype does not have integral values.

13/6

A.5.5 Big Numbers

Support is provided for integer arithmetic involving values larger than those supported by the target machine, and for arbitrary-precision real numbers.

1/5

Static Semantics

The library package Numerics.Big_Numbers has the following declaration:

2/5

```
package Ada.Numerics.Big_Numbers
   with Pure, Nonblocking, Global => null is
   subtype Field is Integer range 0 .. implementation-defined;
   subtype Number_Base is Integer range 2 .. 16;
end Ada.Numerics.Big_Numbers;
```

3/5

A.5.6 Big Integers

Static Semantics

The library package Numerics.Big_Numbers.Big_Integers has the following declaration:

1/5

```
with Ada.Strings.Text_Buffers;
package Ada.Numerics.Big_Numbers.Big_Integers
   with Preelaborate, Nonblocking, Global => in out synchronized is

   type Big_Integer is private
     with Integer_Literal => From_Universal_Image,
          Put_Image => Put_Image;

   function Is_Valid (Arg : Big_Integer) return Boolean
      with Convention => Intrinsic;

   subtype Valid_Big_Integer is Big_Integer
      with Dynamic_Predicate => Is_Valid (Valid_Big_Integer),
           Predicate_Failure => (raise Program_Error);

   function "=" (L, R : Valid_Big_Integer) return Boolean;
   function "<" (L, R : Valid_Big_Integer) return Boolean;
   function "<=" (L, R : Valid_Big_Integer) return Boolean;
   function ">" (L, R : Valid_Big_Integer) return Boolean;
   function ">=" (L, R : Valid_Big_Integer) return Boolean;

   function To_Big_Integer (Arg : Integer) return Valid_Big_Integer;

   subtype Big_Positive is Big_Integer
      with Dynamic_Predicate => (if Is_Valid (Big_Positive)
                                 then Big_Positive > 0),
           Predicate_Failure => (raise Constraint_Error);
```

2/5 3/5 4/5 5/5 6/5 7/5 8/5

9/5
```
      subtype Big_Natural is Big_Integer
         with Dynamic_Predicate => (if Is_Valid (Big_Natural)
                                    then Big_Natural >= 0),
              Predicate_Failure => (raise Constraint_Error);
```

10/5
```
      function In_Range (Arg, Low, High : Valid_Big_Integer) return Boolean is
        (Low <= Arg and Arg <= High);
```

11/5
```
      function To_Integer (Arg : Valid_Big_Integer) return Integer
         with Pre => In_Range (Arg,
                               Low  => To_Big_Integer (Integer'First),
                               High => To_Big_Integer (Integer'Last))
                     or else raise Constraint_Error;
```

12/5
```
      generic
         type Int is range <>;
      package Signed_Conversions is
         function To_Big_Integer (Arg : Int) return Valid_Big_Integer;
         function From_Big_Integer (Arg : Valid_Big_Integer) return Int
            with Pre => In_Range (Arg,
                                  Low  => To_Big_Integer (Int'First),
                                  High => To_Big_Integer (Int'Last))
                        or else raise Constraint_Error;
      end Signed_Conversions;
```

13/5
```
      generic
         type Int is mod <>;
      package Unsigned_Conversions is
         function To_Big_Integer (Arg : Int) return Valid_Big_Integer;
         function From_Big_Integer (Arg : Valid_Big_Integer) return Int
            with Pre => In_Range (Arg,
                                  Low  => To_Big_Integer (Int'First),
                                  High => To_Big_Integer (Int'Last))
                        or else raise Constraint_Error;
      end Unsigned_Conversions;
```

14/5
```
      function To_String (Arg : Valid_Big_Integer;
                          Width : Field := 0;
                          Base  : Number_Base := 10) return String
         with Post => To_String'Result'First = 1;
```

15/5
```
      function From_String (Arg : String) return Valid_Big_Integer;
```

16/5
```
      function From_Universal_Image (Arg : String) return Valid_Big_Integer
         renames From_String;
```

17/5
```
      procedure Put_Image
        (Buffer : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
         Arg    : in Valid_Big_Integer);
```

18/5
```
      function "+" (L : Valid_Big_Integer) return Valid_Big_Integer;
      function "-" (L : Valid_Big_Integer) return Valid_Big_Integer;
      function "abs" (L : Valid_Big_Integer) return Valid_Big_Integer;
      function "+" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
      function "-" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
      function "*" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
      function "/" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
      function "mod" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
      function "rem" (L, R : Valid_Big_Integer) return Valid_Big_Integer;
      function "**" (L : Valid_Big_Integer; R : Natural)
         return Valid_Big_Integer;
      function Min (L, R : Valid_Big_Integer) return Valid_Big_Integer;
      function Max (L, R : Valid_Big_Integer) return Valid_Big_Integer;
```

19/5
```
      function Greatest_Common_Divisor
        (L, R : Valid_Big_Integer) return Big_Positive
        with Pre => (L /= 0 and R /= 0) or else raise Constraint_Error;
```

20/5
```
   private
      ... -- not specified by the language
   end Ada.Numerics.Big_Numbers.Big_Integers;
```

To_String and From_String behave analogously to the Put and Get procedures defined in Text_IO.Integer_IO (in particular, with respect to the interpretation of the Width and Base parameters) except that Constraint_Error, not Data_Error, is propagated in error cases and the result of a call to To_String with a Width parameter of 0 and a nonnegative Arg parameter does not include a leading blank. Put_Image calls To_String (passing in the default values for the Width and Base parameters), prepends a leading blank if the argument is nonnegative, and writes the resulting value to the buffer using Text_Buffers.Put. 21/5

The other functions have their usual mathematical meanings. 22/5

The type Big_Integer needs finalization (see 7.6). 23/5

Dynamic Semantics

For purposes of determining whether predicate checks are performed as part of default initialization, the type Big_Integer is considered to have a subcomponent that has a default_expression. 24/5

Implementation Requirements

No storage associated with a Big_Integer object shall be lost upon assignment or scope exit. 25/5

A.5.7 Big Reals

Static Semantics

The library package Numerics.Big_Numbers.Big_Reals has the following declaration: 1/5

```
with Ada.Numerics.Big_Numbers.Big_Integers;                                    2/5
   use all type Big_Integers.Big_Integer;
with Ada.Strings.Text_Buffers;
package Ada.Numerics.Big_Numbers.Big_Reals
   with Preelaborate, Nonblocking, Global => in out synchronized is

   type Big_Real is private                                                    3/5
      with Real_Literal => From_Universal_Image,
           Put_Image => Put_Image;

   function Is_Valid (Arg : Big_Real) return Boolean                           4/5
      with Convention => Intrinsic;

   subtype Valid_Big_Real is Big_Real                                          5/5
      with Dynamic_Predicate => Is_Valid (Valid_Big_Real),
           Predicate_Failure => raise Program_Error;

   function "/" (Num, Den : Big_Integers.Valid_Big_Integer)                    6/5
      return Valid_Big_Real
      with Pre => Den /= 0
                  or else raise Constraint_Error;

   function Numerator                                                          7/5
      (Arg : Valid_Big_Real) return Big_Integers.Valid_Big_Integer
     with Post => (if Arg = 0.0 then Numerator'Result = 0);

   function Denominator (Arg : Valid_Big_Real)                                 8/5
      return Big_Integers.Big_Positive
      with Post =>
        (if Arg = 0.0 then Denominator'Result = 1
         else Big_Integers.Greatest_Common_Divisor
                (Numerator (Arg), Denominator'Result) = 1);

   function To_Big_Real (Arg : Big_Integers.Valid_Big_Integer)                 9/5
      return Valid_Big_Real is (Arg / 1);

   function To_Real (Arg : Integer) return Valid_Big_Real is                   10/5
      (Big_Integers.To_Big_Integer (Arg) / 1);
```

11/5
```
function "=" (L, R : Valid_Big_Real) return Boolean;
function "<" (L, R : Valid_Big_Real) return Boolean;
function "<=" (L, R : Valid_Big_Real) return Boolean;
function ">" (L, R : Valid_Big_Real) return Boolean;
function ">=" (L, R : Valid_Big_Real) return Boolean;
```

12/5
```
function In_Range (Arg, Low, High : Valid_Big_Real) return Boolean is
   (Low <= Arg and Arg <= High);
```

13/5
```
generic
   type Num is digits <>;
package Float_Conversions is
   function To_Big_Real (Arg : Num) return Valid_Big_Real;
   function From_Big_Real (Arg : Valid_Big_Real) return Num
      with Pre => In_Range (Arg,
                            Low  => To_Big_Real (Num'First),
                            High => To_Big_Real (Num'Last))
                  or else (raise Constraint_Error);
end Float_Conversions;
```

14/5
```
generic
   type Num is delta <>;
package Fixed_Conversions is
   function To_Big_Real (Arg : Num) return Valid_Big_Real;
   function From_Big_Real (Arg : Valid_Big_Real) return Num
      with Pre => In_Range (Arg,
                            Low  => To_Big_Real (Num'First),
                            High => To_Big_Real (Num'Last))
                  or else (raise Constraint_Error);
end Fixed_Conversions;
```

14.1/6
```
generic
   type Num is delta <> digits <>;
package Decimal_Conversions is
   function To_Big_Real (Arg : Num) return Valid_Big_Real;
   function From_Big_Real (Arg : Valid_Big_Real) return Num
      with Pre => In_Range (Arg,
                            Low  => To_Big_Real (Num'First),
                            High => To_Big_Real (Num'Last))
                  or else (raise Constraint_Error);
end Decimal_Conversions;
```

15/5
```
function To_String (Arg  : Valid_Big_Real;
                    Fore : Field := 2;
                    Aft  : Field := 3;
                    Exp  : Field := 0) return String
   with Post => To_String'Result'First = 1;
```

16/5
```
function From_String (Arg   : String) return Valid_Big_Real;
```

17/5
```
function From_Universal_Image (Arg : String) return Valid_Big_Real
   renames From_String;
```

18/5
```
function From_Universal_Image (Num, Den : String)
   return Valid_Big_Real is
      (Big_Integers.From_Universal_Image (Num) /
       Big_Integers.From_Universal_Image (Den));
```

19/5
```
function To_Quotient_String (Arg : Valid_Big_Real) return String is
   (To_String (Numerator (Arg)) & " / " & To_String (Denominator (Arg)));
function From_Quotient_String (Arg : String) return Valid_Big_Real;
```

20/5
```
procedure Put_Image
  (Buffer : in out Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
   Arg    : in Valid_Big_Real);
```

```
      function "+" (L : Valid_Big_Real) return Valid_Big_Real;                    21/5
      function "-" (L : Valid_Big_Real) return Valid_Big_Real;
      function "abs" (L : Valid_Big_Real) return Valid_Big_Real;
      function "+" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "-" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "*" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "/" (L, R : Valid_Big_Real) return Valid_Big_Real;
      function "**" (L : Valid_Big_Real; R : Integer)
         return Valid_Big_Real;
      function Min (L, R : Valid_Big_Real) return Valid_Big_Real;
      function Max (L, R : Valid_Big_Real) return Valid_Big_Real;
   private                                                                        22/5
      ... -- not specified by the language
   end Ada.Numerics.Big_Numbers.Big_Reals;
```

To_String and From_String behave analogously to the Put and Get procedures defined in Text_IO.Float_IO (in particular, with respect to the interpretation of the Fore, Aft, and Exp parameters), except that Constraint_Error (not Data_Error) is propagated in error cases. From_Quotient_String implements the inverse function of To_Quotient_String; Constraint_Error is propagated in error cases. Put_Image calls To_String, and writes the resulting value to the buffer using Text_Buffers.Put. 23/5

For an instance of Float_Conversions, Fixed_Conversions, or Decimal_Conversions, To_Big_Real is exact (that is, the result represents exactly the same mathematical value as the argument) and From_Big_Real is subject to the same precision rules as a type conversion of a value of type T to the target type Num, where T is a hypothetical floating point type whose model numbers include all of the model numbers of Num as well as the exact mathematical value of the argument. 24/6

The other functions have their usual mathematical meanings. 25/5

The type Big_Real needs finalization (see 7.6). 26/5

Dynamic Semantics

For purposes of determining whether predicate checks are performed as part of default initialization, the type Big_Real is considered to have a subcomponent that has a default_expression. 27/5

Implementation Requirements

No storage associated with a Big_Real object shall be lost upon assignment or scope exit. 28/5

A.6 Input-Output

Input-output is provided through language-defined packages, each of which is a child of the root package Ada. The generic packages Sequential_IO and Direct_IO define input-output operations applicable to files containing elements of a given type. The generic package Storage_IO supports reading from and writing to an in-memory buffer. Additional operations for text input-output are supplied in the packages Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. Heterogeneous input-output is provided through the child packages Streams.Stream_IO and Text_IO.Text_Streams (see also 13.13). The package IO_Exceptions defines the exceptions used by the predefined input-output packages. 1/5

A.7 External Files and File Objects

Static Semantics

Values input from the external environment of the program, or output to the external environment, are considered to occupy *external files*. An external file can be anything external to the program that can 1

produce a value to be read or receive a value to be written. An external file is identified by a string (the *name*). A second string (the *form*) gives further system-dependent characteristics that may be associated with the file, such as the physical organization or access rights. The conventions governing the interpretation of such strings shall be documented.

2/3 Input and output operations are expressed as operations on objects of some *file type*, rather than directly in terms of the external files. In the remainder of this clause, the term *file* is always used to refer to a file object; the term *external file* is used otherwise.

3 Input-output for sequential files of values of a single element type is defined by means of the generic package Sequential_IO. In order to define sequential input-output for a given element type, an instantiation of this generic unit, with the given type as actual parameter, has to be declared. The resulting package contains the declaration of a file type (called File_Type) for files of such elements, as well as the operations applicable to these files, such as the Open, Read, and Write procedures.

4/2 Input-output for direct access files is likewise defined by a generic package called Direct_IO. Input-output in human-readable form is defined by the (nongeneric) packages Text_IO for Character and String data, Wide_Text_IO for Wide_Character and Wide_String data, and Wide_Wide_Text_IO for Wide_Wide_Character and Wide_Wide_String data. Input-output for files containing streams of elements representing values of possibly different types is defined by means of the (nongeneric) package Streams.Stream_IO.

5 Before input or output operations can be performed on a file, the file first has to be associated with an external file. While such an association is in effect, the file is said to be *open*, and otherwise the file is said to be *closed*.

6/6 The language does not define what happens to external files after the completion of the main subprogram and all the library tasks (in particular, if corresponding files have not been closed). The effect of input-output for access types is unspecified.

7 An open file has a *current mode*, which is a value of one of the following enumeration types:

8
```
type File_Mode is (In_File, Inout_File, Out_File);  -- for Direct_IO
```

9 These values correspond respectively to the cases where only reading, both reading and writing, or only writing are to be performed.

10/2
```
type File_Mode is (In_File, Out_File, Append_File);
-- for Sequential_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Stream_IO
```

11 These values correspond respectively to the cases where only reading, only writing, or only appending are to be performed.

12 The mode of a file can be changed.

13/5 Several file management operations are common to Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, and Wide_Wide_Text_IO. These operations are described in A.8.2 for sequential and direct files. Any additional effects concerning text input-output are described in A.10.2.

14/5 The exceptions that can be propagated by the execution of an input-output subprogram are defined in the package IO_Exceptions; the situations in which they can be propagated are described following the description of the subprogram (and in A.13). The exceptions Storage_Error and Program_Error may be propagated. (Program_Error can only be propagated due to errors made by the caller of the subprogram.) Finally, exceptions can be propagated in certain implementation-defined situations.

NOTE 1 Each instantiation of the generic packages Sequential_IO and Direct_IO declares a different type File_Type. In the case of Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, and Streams.Stream_IO, the corresponding type File_Type is unique. 15/2

NOTE 2 A bidirectional device can often be modeled as two sequential files associated with the device, one of mode In_File, and one of mode Out_File. An implementation can restrict the number of files that can be associated with a given external file. 16/5

A.8 Sequential and Direct Files

Static Semantics

Two kinds of access to external files are defined in this subclause: *sequential access* and *direct access*. The corresponding file types and the associated operations are provided by the generic packages Sequential_IO and Direct_IO. A file object to be used for sequential access is called a *sequential file*, and one to be used for direct access is called a *direct file*. Access to *stream file*s is described in A.12.1. 1/2

For sequential access, the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or by the external environment). When the file is opened with mode In_File or Out_File, transfer starts respectively from or to the beginning of the file. When the file is opened with mode Append_File, transfer to the file starts after the last element of the file. 2

For direct access, the file is viewed as a set of elements occupying consecutive positions in linear order; a value can be transferred to or from an element of the file at any selected position. The position of an element is specified by its *index*, which is a number, greater than zero, of the implementation-defined integer type Count. The first element, if any, has index one; the index of the last element, if any, is called the *current size*; the current size is zero if there are no elements. The current size is a property of the external file. 3

An open direct file has a *current index*, which is the index that will be used by the next read or write operation. When a direct file is opened, the current index is set to one. The current index of a direct file is a property of a file object, not of an external file. 4

A.8.1 The Generic Package Sequential_IO

Static Semantics

The generic library package Sequential_IO has the following declaration: 1

```
with Ada.IO_Exceptions;                                                    2/5
generic
   type Element_Type(<>) is private;
package Ada.Sequential_IO
   with Global => in out synchronized is

   type File_Type is limited private;                                      3

   type File_Mode is (In_File, Out_File, Append_File);                     4

   -- File management                                                      5

   procedure Create(File : in out File_Type;                               6
                    Mode : in File_Mode := Out_File;
                    Name : in String := "";
                    Form : in String := "");

   procedure Open  (File : in out File_Type;                               7
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");
```

```
8           procedure Close (File : in out File_Type);
            procedure Delete(File : in out File_Type);
            procedure Reset (File : in out File_Type; Mode : in File_Mode);
            procedure Reset (File : in out File_Type);
9           function Mode   (File : in File_Type) return File_Mode;
            function Name   (File : in File_Type) return String;
            function Form   (File : in File_Type) return String;
10          function Is_Open(File : in File_Type) return Boolean;
10.1/5      procedure Flush (File : in File_Type)
                with Global => overriding in out File;
11          -- Input and output operations
12/5        procedure Read  (File : in File_Type; Item : out Element_Type)
                with Global => overriding in out File;
            procedure Write (File : in File_Type; Item : in Element_Type)
                with Global => overriding in out File;
13          function End_Of_File(File : in File_Type) return Boolean;
14          -- Exceptions
15          Status_Error : exception renames IO_Exceptions.Status_Error;
            Mode_Error   : exception renames IO_Exceptions.Mode_Error;
            Name_Error   : exception renames IO_Exceptions.Name_Error;
            Use_Error    : exception renames IO_Exceptions.Use_Error;
            Device_Error : exception renames IO_Exceptions.Device_Error;
            End_Error    : exception renames IO_Exceptions.End_Error;
            Data_Error   : exception renames IO_Exceptions.Data_Error;
15.1/5      package Wide_File_Names is
15.2/5         -- File management
15.3/5         procedure Create(File : in out File_Type;
                                Mode : in File_Mode := Out_File;
                                Name : in Wide_String := "";
                                Form : in Wide_String := "");
15.4/5         procedure Open  (File : in out File_Type;
                                Mode : in File_Mode;
                                Name : in Wide_String;
                                Form : in Wide_String := "");
15.5/5         function Name   (File : in File_Type) return Wide_String;
15.6/5         function Form   (File : in File_Type) return Wide_String;
15.7/5      end Wide_File_Names;
15.8/5      package Wide_Wide_File_Names is
15.9/5         -- File management
15.10/5        procedure Create(File : in out File_Type;
                                Mode : in File_Mode := Out_File;
                                Name : in Wide_Wide_String := "";
                                Form : in Wide_Wide_String := "");
15.11/5        procedure Open  (File : in out File_Type;
                                Mode : in File_Mode;
                                Name : in Wide_Wide_String;
                                Form : in Wide_Wide_String := "");
15.12/5        function Name   (File : in File_Type) return Wide_Wide_String;
15.13/5        function Form   (File : in File_Type) return Wide_Wide_String;
15.14/5     end Wide_Wide_File_Names;
16       private
            ... -- not specified by the language
         end Ada.Sequential_IO;
```

17/2 The type File_Type needs finalization (see 7.6) in every instantiation of Sequential_IO.

A.8.2 File Management

Static Semantics

The procedures and functions described in this subclause provide for the control of external files; their declarations are repeated in each of the packages for sequential, direct, text, and stream input-output. For text input-output, the procedures Create, Open, and Reset have additional effects described in A.10.2. 1/5

```
procedure Create(File : in out File_Type;                                    2
                 Mode : in File_Mode := default_mode;
                 Name : in String := "";
                 Form : in String := "");
```

Establishes a new external file, with the given name and form, and associates this external file with the given file. The given file is left open. The current mode of the given file is set to the given access mode. The default access mode is the mode Out_File for sequential, stream, and text input-output; it is the mode Inout_File for direct input-output. For direct access, the size of the created file is implementation defined. 3/2

A null string for Name specifies an external file that is not accessible after the completion of the partition (a temporary file). A null string for Form specifies the use of the default options of the implementation for the external file. 4/6

The exception Status_Error is propagated if the given file is already open. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if, for the specified mode, the external environment does not support creation of an external file with the given name (in the absence of Name_Error) and form. 5

```
procedure Open(File : in out File_Type;                                      6
               Mode : in File_Mode;
               Name : in String;
               Form : in String := "");
```

Associates the given file with an existing external file having the given name and form, and sets the current mode of the given file to the given mode. The given file is left open. 7

The exception Status_Error is propagated if the given file is already open. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file; in particular, this exception is propagated if no external file with the given name exists. The exception Use_Error is propagated if, for the specified mode, the external environment does not support opening for an external file with the given name (in the absence of Name_Error) and form. 8

```
procedure Close(File : in out File_Type);                                    9
```

Severs the association between the given file and its associated external file. The given file is left closed. In addition, for sequential files, if the file being closed has mode Out_File or Append_File, then the last element written since the most recent open or reset is the last element that can be read from the file. If no elements have been written and the file mode is Out_File, then the closed file is empty. If no elements have been written and the file mode is Append_File, then the closed file is unchanged. 10

The exception Status_Error is propagated if the given file is not open. 11

12 `procedure Delete(File : in out File_Type);`

13 Deletes the external file associated with the given file. The given file is closed, and the external file ceases to exist.

14 The exception Status_Error is propagated if the given file is not open. The exception Use_Error is propagated if deletion of the external file is not supported by the external environment.

15 `procedure Reset(File : in out File_Type; Mode : in File_Mode);`
`procedure Reset(File : in out File_Type);`

16/2 Resets the given file so that reading from its elements can be restarted from the beginning of the external file (for modes In_File and Inout_File), and so that writing to its elements can be restarted at the beginning of the external file (for modes Out_File and Inout_File) or after the last element of the external file (for mode Append_File). In particular, for direct access this means that the current index is set to one. If a Mode parameter is supplied, the current mode of the given file is set to the given mode. In addition, for sequential files, if the given file has mode Out_File or Append_File when Reset is called, the last element written since the most recent open or reset is the last element that can be read from the external file. If no elements have been written and the file mode is Out_File, the reset file is empty. If no elements have been written and the file mode is Append_File, then the reset file is unchanged.

17 The exception Status_Error is propagated if the file is not open. The exception Use_Error is propagated if the external environment does not support resetting for the external file and, also, if the external environment does not support resetting to the specified mode for the external file.

18 `function Mode(File : in File_Type) return File_Mode;`

19 Returns the current mode of the given file.

20 The exception Status_Error is propagated if the file is not open.

21 `function Name(File : in File_Type) return String;`

22/2 Returns a string which uniquely identifies the external file currently associated with the given file (and may thus be used in an Open operation).

23 The exception Status_Error is propagated if the given file is not open. The exception Use_Error is propagated if the associated external file is a temporary file that cannot be opened by any name.

24 `function Form(File : in File_Type) return String;`

25 Returns the form string for the external file currently associated with the given file. If an external environment allows alternative specifications of the form (for example, abbreviations using default options), the string returned by the function should correspond to a full specification (that is, it should indicate explicitly all options selected, including default options).

26 The exception Status_Error is propagated if the given file is not open.

27 `function Is_Open(File : in File_Type) return Boolean;`

28/3 Returns True if the file is open (that is, if it is associated with an external file); otherwise, returns False.

28.1/4 `procedure Flush(File : in File_Type);`

The Flush procedure synchronizes the external file with the internal file (by flushing any internal buffers) without closing the file. For a direct file, the current index is unchanged; for a stream file (see A.12.1), the current position is unchanged. 28.2/4

The exception Status_Error is propagated if the file is not open. The exception Mode_Error is propagated if the mode of the file is In_File. 28.3/4

The nested package Wide_File_Names provides operations equivalent to the operations of the same name of the outer package except that Wide_String is used instead of String for the name and form of the external file. 28.4/5

The nested package Wide_Wide_File_Names provides operations equivalent to the operations of the same name of the outer package except that Wide_Wide_String is used instead of String for the name and form of the external file. 28.5/5

Implementation Permissions

An implementation may propagate Name_Error or Use_Error if an attempt is made to use an I/O feature that cannot be supported by the implementation due to limitations in the external environment. Any such restriction should be documented. 29

A.8.3 Sequential Input-Output Operations

Static Semantics

The operations available for sequential input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open. 1

```
procedure Read(File : in File_Type; Item : out Element_Type);
```
2

Operates on a file of mode In_File. Reads an element from the given file, and returns the value of this element in the Item parameter. 3

The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if no more elements can be read from the given file. The exception Data_Error can be propagated if the element read cannot be interpreted as a value of the subtype Element_Type (see A.13, “Exceptions in Input-Output”). 4

```
procedure Write(File : in File_Type; Item : in Element_Type);
```
5

Operates on a file of mode Out_File or Append_File. Writes the value of Item to the given file. 6

The exception Mode_Error is propagated if the mode is not Out_File or Append_File. The exception Use_Error is propagated if the capacity of the external file is exceeded. 7

```
function End_Of_File(File : in File_Type) return Boolean;
```
8

Operates on a file of mode In_File. Returns True if no more elements can be read from the given file; otherwise, returns False. 9/3

The exception Mode_Error is propagated if the mode is not In_File. 10

A.8.4 The Generic Package Direct_IO

Static Semantics

1 The generic library package Direct_IO has the following declaration:

2/5
```
with Ada.IO_Exceptions;
generic
   type Element_Type is private;
package Ada.Direct_IO
   with Global => in out synchronized is
```

3
```
   type File_Type is limited private;
```

4
```
   type File_Mode is (In_File, Inout_File, Out_File);
   type Count     is range 0 .. implementation-defined;
   subtype Positive_Count is Count range 1 .. Count'Last;
```

5
```
   -- File management
```

6
```
   procedure Create(File : in out File_Type;
                    Mode : in File_Mode := Inout_File;
                    Name : in String := "";
                    Form : in String := "");
```

7
```
   procedure Open  (File : in out File_Type;
                    Mode : in File_Mode;
                    Name : in String;
                    Form : in String := "");
```

8
```
   procedure Close (File : in out File_Type);
   procedure Delete(File : in out File_Type);
   procedure Reset (File : in out File_Type; Mode : in File_Mode);
   procedure Reset (File : in out File_Type);
```

9
```
   function Mode   (File : in File_Type) return File_Mode;
   function Name   (File : in File_Type) return String;
   function Form   (File : in File_Type) return String;
```

10
```
   function Is_Open(File : in File_Type) return Boolean;
```

10.1/5
```
   procedure Flush (File : in File_Type)
      with Global => overriding in out File;
```

11
```
   -- Input and output operations
```

12/5
```
   procedure Read (File : in File_Type; Item : out Element_Type;
                                        From : in Positive_Count)
      with Global => overriding in out File;
   procedure Read (File : in File_Type; Item : out Element_Type)
      with Global => overriding in out File;
```

13/5
```
   procedure Write(File : in File_Type; Item : in  Element_Type;
                                        To   : in Positive_Count)
      with Global => overriding in out File;
   procedure Write(File : in File_Type; Item : in Element_Type)
      with Global => overriding in out File;
```

14/5
```
   procedure Set_Index(File : in File_Type; To : in Positive_Count)
      with Global => overriding in out File;
```

15
```
   function Index(File : in File_Type) return Positive_Count;
   function Size (File : in File_Type) return Count;
```

16
```
   function End_Of_File(File : in File_Type) return Boolean;
```

```
   -- Exceptions                                                                    17
   Status_Error : exception renames IO_Exceptions.Status_Error;                    18
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;

   package Wide_File_Names is                                                      18.1/5
      -- File management                                                          18.2/5
      procedure Create(File : in out File_Type;                                    18.3/5
                       Mode : in File_Mode := Inout_File;
                       Name : in Wide_String := "";
                       Form : in Wide_String := "");
      procedure Open  (File : in out File_Type;                                    18.4/5
                       Mode : in File_Mode;
                       Name : in Wide_String;
                       Form : in Wide_String := "");
      function Name   (File : in File_Type) return Wide_String;                    18.5/5
      function Form   (File : in File_Type) return Wide_String;                    18.6/5
   end Wide_File_Names;                                                            18.7/5
   package Wide_Wide_File_Names is                                                 18.8/5
      -- File management                                                          18.9/5
      procedure Create(File : in out File_Type;                                    18.10/5
                       Mode : in File_Mode := Inout_File;
                       Name : in Wide_Wide_String := "";
                       Form : in Wide_Wide_String := "");
      procedure Open  (File : in out File_Type;                                    18.11/5
                       Mode : in File_Mode;
                       Name : in Wide_Wide_String;
                       Form : in Wide_Wide_String := "");
      function Name   (File : in File_Type) return Wide_Wide_String;               18.12/5
      function Form   (File : in File_Type) return Wide_Wide_String;               18.13/5
   end Wide_Wide_File_Names;                                                       18.14/5
private                                                                            19
   ... -- not specified by the language
end Ada.Direct_IO;
```

The type File_Type needs finalization (see 7.6) in every instantiation of Direct_IO. 20/2

A.8.5 Direct Input-Output Operations

Static Semantics

The operations available for direct input and output are described in this subclause. The exception Status_Error is propagated if any of these operations is attempted for a file that is not open. 1

```
procedure Read(File : in File_Type; Item : out Element_Type;                       2
                                    From : in  Positive_Count);
procedure Read(File : in File_Type; Item : out Element_Type);
```

Operates on a file of mode In_File or Inout_File. In the case of the first form, sets the current index of the given file to the index value given by the parameter From. Then (for both forms) returns, in the parameter Item, the value of the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one. 3

4 The exception Mode_Error is propagated if the mode of the given file is Out_File. The exception End_Error is propagated if the index to be used exceeds the size of the external file. The exception Data_Error can be propagated if the element read cannot be interpreted as a value of the subtype Element_Type (see A.13).

5
```
procedure Write(File : in File_Type; Item : in Element_Type;
                                     To   : in Positive_Count);
procedure Write(File : in File_Type; Item : in Element_Type);
```

6 Operates on a file of mode Inout_File or Out_File. In the case of the first form, sets the index of the given file to the index value given by the parameter To. Then (for both forms) gives the value of the parameter Item to the element whose position in the given file is specified by the current index of the file; finally, increases the current index by one.

7 The exception Mode_Error is propagated if the mode of the given file is In_File. The exception Use_Error is propagated if the capacity of the external file is exceeded.

8
```
procedure Set_Index(File : in File_Type; To : in Positive_Count);
```

9 Operates on a file of any mode. Sets the current index of the given file to the given index value (which may exceed the current size of the file).

10
```
function Index(File : in File_Type) return Positive_Count;
```

11 Operates on a file of any mode. Returns the current index of the given file.

12
```
function Size(File : in File_Type) return Count;
```

13 Operates on a file of any mode. Returns the current size of the external file that is associated with the given file.

14
```
function End_Of_File(File : in File_Type) return Boolean;
```

15/3 Operates on a file of mode In_File or Inout_File. Returns True if the current index exceeds the size of the external file; otherwise, returns False.

16 The exception Mode_Error is propagated if the mode of the given file is Out_File.

17 NOTE Append_File mode is not supported for the generic package Direct_IO.

A.9 The Generic Package Storage_IO

1 The generic package Storage_IO provides for reading from and writing to an in-memory buffer. This generic package supports the construction of user-defined input-output packages.

Static Semantics

2 The generic library package Storage_IO has the following declaration:

3/5
```
with Ada.IO_Exceptions;
with System.Storage_Elements;
generic
   type Element_Type is private;
package Ada.Storage_IO
   with Preelaborate, Global => in out synchronized is

4     Buffer_Size : constant System.Storage_Elements.Storage_Count :=
         implementation-defined;
      subtype Buffer_Type is
         System.Storage_Elements.Storage_Array(1..Buffer_Size);

5     -- Input and output operations
```

```
   procedure Read (Buffer : in  Buffer_Type; Item : out Element_Type);
   procedure Write(Buffer : out Buffer_Type; Item : in  Element_Type);
   -- Exceptions
   Data_Error   : exception renames IO_Exceptions.Data_Error;
end Ada.Storage_IO;
```

In each instance, the constant Buffer_Size has a value that is the size (in storage elements) of the buffer required to represent the content of an object of subtype Element_Type, including any implicit levels of indirection used by the implementation. The Read and Write procedures of Storage_IO correspond to the Read and Write procedures of Direct_IO (see A.8.4), but with the content of the Item parameter being read from or written into the specified Buffer, rather than an external file.

NOTE A buffer used for Storage_IO holds only one element at a time; an external file used for Direct_IO holds a sequence of elements.

A.10 Text Input-Output

Static Semantics

This subclause describes the package Text_IO, which provides facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of characters grouped into lines, and as a sequence of lines grouped into pages. The specification of the package is given below in A.10.1.

The facilities for file management given above, in A.8.2 and A.8.3, are available for text input-output. In place of Read and Write, however, there are procedures Get and Put that input values of suitable types from text files, and output values to them. These values are provided to the Put procedures, and returned by the Get procedures, in a parameter Item. Several overloaded procedures of these names exist, for different types of Item. These Get procedures analyze the input sequences of characters based on lexical elements (see Clause 2) and return the corresponding values; the Put procedures output the given values as appropriate lexical elements. Procedures Get and Put are also available that input and output individual characters treated as character values rather than as lexical elements. Related to character input are procedures to look ahead at the next character without reading it, and to read a character "immediately" without waiting for an end-of-line to signal availability.

In addition to the procedures Get and Put for numeric and enumeration types of Item that operate on text files, analogous procedures are provided that read from and write to a parameter of type String. These procedures perform the same analysis and composition of character sequences as their counterparts which have a file parameter.

For all Get and Put procedures that operate on text files, and for many other subprograms, there are forms with and without a file parameter. Each such Get procedure operates on an input file, and each such Put procedure operates on an output file. If no file is specified, a default input file or a default output file is used.

At the beginning of program execution the default input and output files are the so-called standard input file and standard output file. These files are open, have respectively the current modes In_File and Out_File, and are associated with two implementation-defined external files. Procedures are provided to change the current default input file and the current default output file.

At the beginning of program execution a default file for program-dependent error-related text output is the so-called standard error file. This file is open, has the current mode Out_File, and is associated with an implementation-defined external file. A procedure is provided to change the current default error file.

7 From a logical point of view, a text file is a sequence of pages, a page is a sequence of lines, and a line is a sequence of characters; the end of a line is marked by a *line terminator*; the end of a page is marked by the combination of a line terminator immediately followed by a *page terminator*; and the end of a file is marked by the combination of a line terminator immediately followed by a page terminator and then a *file terminator*. Terminators are generated during output; either by calls of procedures provided expressly for that purpose; or implicitly as part of other operations, for example, when a bounded line length, a bounded page length, or both, have been specified for a file.

8/5 The actual nature of terminators is not defined by the language and hence depends on the implementation. Although terminators are recognized or generated by certain of the procedures that follow, they are not necessarily implemented as characters or as sequences of characters. Whether they are characters (and if so which ones) in any particular implementation is not of concern to a user who neither explicitly outputs nor explicitly inputs control characters. The effect of input (Get) or output (Put) of control characters (other than horizontal tabulation) is not specified by the language.

9 The characters of a line are numbered, starting from one; the number of a character is called its *column number*. For a line terminator, a column number is also defined: it is one more than the number of characters in the line. The lines of a page, and the pages of a file, are similarly numbered. The current column number is the column number of the next character or line terminator to be transferred. The current line number is the number of the current line. The current page number is the number of the current page. These numbers are values of the subtype Positive_Count of the type Count (by convention, the value zero of the type Count is used to indicate special conditions).

10
```
type Count is range 0 .. implementation-defined;
subtype Positive_Count is Count range 1 .. Count'Last;
```

11 For an output file or an append file, a *maximum line length* can be specified and a *maximum page length* can be specified. If a value to be output cannot fit on the current line, for a specified maximum line length, then a new line is automatically started before the value is output; if, further, this new line cannot fit on the current page, for a specified maximum page length, then a new page is automatically started before the value is output. Functions are provided to determine the maximum line length and the maximum page length. When a file is opened with mode Out_File or Append_File, both values are zero: by convention, this means that the line lengths and page lengths are unbounded. (Consequently, output consists of a single line if the subprograms for explicit control of line and page structure are not used.) The constant Unbounded is provided for this purpose.

A.10.1 The Package Text_IO

Static Semantics

1 The library package Text_IO has the following declaration:

2/5
```
with Ada.IO_Exceptions;
package Ada.Text_IO
   with Global => in out synchronized is

3     type File_Type is limited private;

4     type File_Mode is (In_File, Out_File, Append_File);

5     type Count is range 0 .. implementation-defined;
      subtype Positive_Count is Count range 1 .. Count'Last;
      Unbounded : constant Count := 0; -- line and page length

6     subtype Field       is Integer range 0 .. implementation-defined;
      subtype Number_Base is Integer range 2 .. 16;

7     type Type_Set is (Lower_Case, Upper_Case);
```

```
   -- File Management                                                                    8
   procedure Create (File : in out File_Type;                                           9
                     Mode : in File_Mode := Out_File;
                     Name : in String    := "";
                     Form : in String    := "");
   procedure Open   (File : in out File_Type;                                           10
                     Mode : in File_Mode;
                     Name : in String;
                     Form : in String := "");
   procedure Close  (File : in out File_Type);                                          11
   procedure Delete (File : in out File_Type);
   procedure Reset  (File : in out File_Type; Mode : in File_Mode);
   procedure Reset  (File : in out File_Type);
   function  Mode   (File : in File_Type) return File_Mode;                             12
   function  Name   (File : in File_Type) return String;
   function  Form   (File : in File_Type) return String;
   function  Is_Open(File : in File_Type) return Boolean;                               13
   -- Control of default input and output files                                         14
   procedure Set_Input (File : in File_Type);                                           15
   procedure Set_Output(File : in File_Type);
   procedure Set_Error (File : in File_Type);
   function Standard_Input  return File_Type;                                           16
   function Standard_Output return File_Type;
   function Standard_Error  return File_Type;
   function Current_Input   return File_Type;                                           17
   function Current_Output  return File_Type;
   function Current_Error   return File_Type;
   type File_Access is access constant File_Type;                                       18
   function Standard_Input  return File_Access;                                         19
   function Standard_Output return File_Access;
   function Standard_Error  return File_Access;
   function Current_Input   return File_Access;                                         20
   function Current_Output  return File_Access;
   function Current_Error   return File_Access;
--Buffer control                                                                         21/5
   procedure Flush (File : in File_Type)
      with Global => overriding in out File;
   procedure Flush
      with Global => in out all;
   -- Specification of line and page lengths                                            22
   procedure Set_Line_Length(File : in File_Type; To : in Count)                        23/5
      with Global => overriding in out File;
   procedure Set_Line_Length(To   : in Count)
      with Global => in out all;
   procedure Set_Page_Length(File : in File_Type; To : in Count)                        24/5
      with Global => overriding in out File;
   procedure Set_Page_Length(To   : in Count)
      with Global => in out all;
   function  Line_Length(File : in File_Type) return Count;                             25/5
   function  Line_Length return Count
      with Global => in all;
   function  Page_Length(File : in File_Type) return Count;                             26/5
   function  Page_Length return Count
      with Global => in all;
   -- Column, Line, and Page Control                                                    27
```

```
28/5        procedure New_Line   (File    : in File_Type;
                                  Spacing : in Positive_Count := 1)
               with Global => overriding in out File;
            procedure New_Line   (Spacing : in Positive_Count := 1)
               with Global => in out all;
29/5        procedure Skip_Line  (File    : in File_Type;
                                  Spacing : in Positive_Count := 1)
               with Global => overriding in out File;
            procedure Skip_Line  (Spacing : in Positive_Count := 1)
               with Global => in out all;
30/5        function  End_Of_Line(File : in File_Type) return Boolean;
            function  End_Of_Line return Boolean;
31/5        procedure New_Page   (File : in File_Type)
               with Global => overriding in out File;
            procedure New_Page
               with Global => in out all;
32/5        procedure Skip_Page  (File : in File_Type)
               with Global => overriding in out File;
            procedure Skip_Page
               with Global => in out all;
33/5        function  End_Of_Page(File : in File_Type) return Boolean;
            function  End_Of_Page return Boolean
               with Global => in all;
34/5        function  End_Of_File(File : in File_Type) return Boolean;
            function  End_Of_File return Boolean
               with Global => in all;
35/5        procedure Set_Col (File : in File_Type; To : in Positive_Count)
               with Global => overriding in out File;
            procedure Set_Col (To   : in Positive_Count)
               with Global => in out all;
36/5        procedure Set_Line(File : in File_Type; To : in Positive_Count)
               with Global => overriding in out File;
            procedure Set_Line(To   : in Positive_Count)
               with Global => in out all;
37/5        function Col (File : in File_Type) return Positive_Count;
            function Col  return Positive_Count
               with Global => in all;
38/5        function Line(File : in File_Type) return Positive_Count;
            function Line return Positive_Count
               with Global => in all;
39/5        function Page(File : in File_Type) return Positive_Count;
            function Page return Positive_Count
               with Global => in all;
40          -- Character Input-Output
41/5        procedure Get(File : in  File_Type; Item : out Character)
               with Global => overriding in out File;
            procedure Get(Item : out Character)
               with Global => in out all;
42/5        procedure Put(File : in  File_Type; Item : in Character)
               with Global => overriding in out File;
            procedure Put(Item : in  Character)
               with Global => in out all;
43/5        procedure Look_Ahead (File        : in  File_Type;
                                  Item        : out Character;
                                  End_Of_Line : out Boolean)
               with Global => overriding in out File;
            procedure Look_Ahead (Item        : out Character;
                                  End_Of_Line : out Boolean)
               with Global => in out all;
```

```
   procedure Get_Immediate(File      : in  File_Type;                    44/5
                           Item      : out Character)
      with Global => overriding in out File;
   procedure Get_Immediate(Item      : out Character)
      with Global => in out all;

   procedure Get_Immediate(File      : in  File_Type;                    45/5
                           Item      : out Character;
                           Available : out Boolean)
      with Global => overriding in out File;
   procedure Get_Immediate(Item      : out Character;
                           Available : out Boolean)
      with Global => in out all;

   -- String Input-Output                                                46

   procedure Get(File : in  File_Type; Item : out String)                47/5
      with Global => overriding in out File;
   procedure Get(Item : out String)
      with Global => in out all;

   procedure Put(File : in  File_Type; Item : in String)                 48/5
      with Global => overriding in out File;
   procedure Put(Item : in  String)
      with Global => in out all;

   procedure Get_Line(File : in  File_Type;                              49/5
                      Item : out String;
                      Last : out Natural)
      with Global => overriding in out File;
   procedure Get_Line(Item : out String; Last : out Natural)
      with Global => in out all;

   function Get_Line(File : in  File_Type) return String                 49.1/5
      with Global => overriding in out File;
   function Get_Line return String
      with Global => in out all;

   procedure Put_Line(File : in  File_Type; Item : in String)            50/5
      with Global => overriding in out File;
   procedure Put_Line(Item : in  String)
      with Global => in out all;

-- Generic packages for Input-Output of Integer Types                    51

   generic                                                               52
      type Num is range <>;
   package Integer_IO is

      Default_Width : Field := Num'Width;                                53
      Default_Base  : Number_Base := 10;

      procedure Get(File  : in  File_Type;                               54/5
                    Item  : out Num;
                    Width : in Field := 0)
         with Global => overriding in out File;
      procedure Get(Item  : out Num;
                    Width : in  Field := 0)
         with Global => in out all;
```

```
55/5        procedure Put(File  : in File_Type;
                          Item  : in Num;
                          Width : in Field := Default_Width;
                          Base  : in Number_Base := Default_Base)
               with Global => overriding in out File;
            procedure Put(Item  : in Num;
                          Width : in Field := Default_Width;
                          Base  : in Number_Base := Default_Base)
               with Global => in out all;
            procedure Get(From : in  String;
                          Item : out Num;
                          Last : out Positive)
               with Nonblocking;
            procedure Put(To   : out String;
                          Item : in Num;
                          Base : in Number_Base := Default_Base)
               with Nonblocking;
56       end Integer_IO;
57       generic
            type Num is mod <>;
         package Modular_IO is
58          Default_Width : Field := Num'Width;
            Default_Base  : Number_Base := 10;
59/5        procedure Get(File  : in  File_Type;
                          Item  : out Num;
                          Width : in Field := 0)
               with Global => overriding in out File;
            procedure Get(Item  : out Num;
                          Width : in  Field := 0)
               with Global => in out all;
60/5        procedure Put(File  : in File_Type;
                          Item  : in Num;
                          Width : in Field := Default_Width;
                          Base  : in Number_Base := Default_Base)
               with Global => overriding in out File;
            procedure Put(Item  : in Num;
                          Width : in Field := Default_Width;
                          Base  : in Number_Base := Default_Base)
               with Global => in out all;
            procedure Get(From : in  String;
                          Item : out Num;
                          Last : out Positive)
               with Nonblocking;
            procedure Put(To   : out String;
                          Item : in Num;
                          Base : in Number_Base := Default_Base)
               with Nonblocking;
61       end Modular_IO;
62       -- Generic packages for Input-Output of Real Types
63       generic
            type Num is digits <>;
         package Float_IO is
64          Default_Fore : Field := 2;
            Default_Aft  : Field := Num'Digits-1;
            Default_Exp  : Field := 3;
65/5        procedure Get(File  : in  File_Type;
                          Item  : out Num;
                          Width : in  Field := 0)
               with Global => overriding in out File;
            procedure Get(Item  : out Num;
                          Width : in  Field := 0)
               with Global => in out all;
```

```
   procedure Put(File : in File_Type;                                    66/5
                 Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp)
      with Global => overriding in out File;
   procedure Put(Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp)
      with Global => in out all;

   procedure Get(From : in String;                                       67/5
                 Item : out Num;
                 Last : out Positive)
      with Nonblocking;
   procedure Put(To   : out String;
                 Item : in Num;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp)
      with Nonblocking;
end Float_IO;

generic                                                                  68
   type Num is delta <>;
package Fixed_IO is

   Default_Fore : Field := Num'Fore;                                      69
   Default_Aft  : Field := Num'Aft;
   Default_Exp  : Field := 0;

   procedure Get(File  : in  File_Type;                                  70/5
                 Item  : out Num;
                 Width : in  Field := 0)
      with Global => overriding in out File;
   procedure Get(Item  : out Num;
                 Width : in  Field := 0)
      with Global => in out all;

   procedure Put(File : in File_Type;                                    71/5
                 Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp)
      with Global => overriding in out File;
   procedure Put(Item : in Num;
                 Fore : in Field := Default_Fore;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp)
      with Global => in out all;

   procedure Get(From : in  String;                                      72/5
                 Item : out Num;
                 Last : out Positive)
      with Nonblocking;
   procedure Put(To   : out String;
                 Item : in Num;
                 Aft  : in Field := Default_Aft;
                 Exp  : in Field := Default_Exp)
      with Nonblocking;
end Fixed_IO;

generic                                                                  73
   type Num is delta <> digits <>;
package Decimal_IO is

   Default_Fore : Field := Num'Fore;                                      74
   Default_Aft  : Field := Num'Aft;
   Default_Exp  : Field := 0;
```

```
75/5           procedure Get(File  : in  File_Type;
                             Item  : out Num;
                             Width : in  Field := 0)
                  with Global => overriding in out File;
               procedure Get(Item  : out Num;
                             Width : in  Field := 0)
                  with Global => in out all;
76/5           procedure Put(File : in File_Type;
                             Item : in Num;
                             Fore : in Field := Default_Fore;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                  with Global => overriding in out File;
               procedure Put(Item : in Num;
                             Fore : in Field := Default_Fore;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                  with Global => in out all;
77/5           procedure Get(From : in  String;
                             Item : out Num;
                             Last : out Positive)
                  with Nonblocking;
               procedure Put(To   : out String;
                             Item : in Num;
                             Aft  : in Field := Default_Aft;
                             Exp  : in Field := Default_Exp)
                  with Nonblocking;
            end Decimal_IO;
78          -- Generic package for Input-Output of Enumeration Types
79          generic
               type Enum is (<>);
            package Enumeration_IO is
80             Default_Width   : Field := 0;
               Default_Setting : Type_Set := Upper_Case;
81/5           procedure Get(File : in  File_Type;
                             Item : out Enum)
                  with Global => overriding in out File;
               procedure Get(Item : out Enum)
                  with Global => in out all;
82/5           procedure Put(File  : in File_Type;
                             Item  : in Enum;
                             Width : in Field    := Default_Width;
                             Set   : in Type_Set := Default_Setting)
                  with Global => overriding in out File;
               procedure Put(Item  : in Enum;
                             Width : in Field    := Default_Width;
                             Set   : in Type_Set := Default_Setting)
                  with Global => in out all;
83/5           procedure Get(From : in  String;
                             Item : out Enum;
                             Last : out Positive)
                  with Nonblocking;
               procedure Put(To   : out String;
                             Item : in  Enum;
                             Set  : in  Type_Set := Default_Setting)
                  with Nonblocking;
            end Enumeration_IO;
```

```
   -- Exceptions                                                                    84
      Status_Error : exception renames IO_Exceptions.Status_Error;                 85/5
      Mode_Error   : exception renames IO_Exceptions.Mode_Error;
      Name_Error   : exception renames IO_Exceptions.Name_Error;
      Use_Error    : exception renames IO_Exceptions.Use_Error;
      Device_Error : exception renames IO_Exceptions.Device_Error;
      End_Error    : exception renames IO_Exceptions.End_Error;
      Data_Error   : exception renames IO_Exceptions.Data_Error;
      Layout_Error : exception renames IO_Exceptions.Layout_Error;

      package Wide_File_Names is                                                   85.1/5
         -- File management                                                        85.2/5
         procedure Create (File : in out File_Type;                                85.3/5
                           Mode : in File_Mode := Out_File;
                           Name : in Wide_String := "";
                           Form : in Wide_String := "");
         procedure Open   (File : in out File_Type;                                85.4/5
                           Mode : in File_Mode;
                           Name : in Wide_String;
                           Form : in Wide_String := "");
         function Name    (File : in File_Type) return Wide_String;               85.5/5
         function Form    (File : in File_Type) return Wide_String;               85.6/5
      end Wide_File_Names;                                                         85.7/5
      package Wide_Wide_File_Names is                                              85.8/5
         -- File management                                                        85.9/5
         procedure Create (File : in out File_Type;                                85.10/5
                           Mode : in File_Mode := Out_File;
                           Name : in Wide_Wide_String := "";
                           Form : in Wide_Wide_String := "");
         procedure Open   (File : in out File_Type;                                85.11/5
                           Mode : in File_Mode;
                           Name : in Wide_Wide_String;
                           Form : in Wide_Wide_String := "");
         function Name    (File : in File_Type) return Wide_Wide_String;          85.12/5
         function Form    (File : in File_Type) return Wide_Wide_String;          85.13/5
      end Wide_Wide_File_Names;                                                    85.14/5
   private                                                                         85.15/5
      ... -- not specified by the language
   end Ada.Text_IO;
```

The type File_Type needs finalization (see 7.6). 86/2

A.10.2 Text File Management

Static Semantics

The only allowed file modes for text files are the modes In_File, Out_File, and Append_File. The subprograms given in A.8.2 for the control of external files, and the function End_Of_File given in A.8.3 for sequential input-output, are also available for text files. There is also a version of End_Of_File that refers to the current default input file. For text files, the procedures have the following additional effects: 1/5

- For the procedures Create and Open: After a file with mode Out_File or Append_File is opened, the page length and line length are unbounded (both have the conventional value zero). After a file (of any mode) is opened, the current column, current line, and current page numbers are set to one. If the mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written. 2

- For the procedure Close: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator.
- For the procedure Reset: If the file has the current mode Out_File or Append_File, has the effect of calling New_Page, unless the current page is already terminated; then outputs a file terminator. The current column, line, and page numbers are set to one, and the line and page lengths to Unbounded. If the new mode is Append_File, it is implementation defined whether a page terminator will separate preexisting text in the file from the new text to be written.

The exception Mode_Error is propagated by the procedure Reset upon an attempt to change the mode of a file that is the current default input file, the current default output file, or the current default error file.

NOTE An implementation can define the Form parameter of Create and Open to control effects including the following:

- the interpretation of line and column numbers for an interactive file, and
- the interpretation of text formats in a file created by a foreign program.

A.10.3 Default Input, Output, and Error Files

Static Semantics

The following subprograms provide for the control of the particular default files that are used when a file parameter is omitted from a Get, Put, or other operation of text input-output described below, or when application-dependent error-related text is to be output.

```
procedure Set_Input(File : in File_Type);
```

Operates on a file of mode In_File. Sets the current default input file to File.

The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not In_File.

```
procedure Set_Output(File : in File_Type);
procedure Set_Error (File : in File_Type);
```

Each operates on a file of mode Out_File or Append_File. Set_Output sets the current default output file to File. Set_Error sets the current default error file to File. The exception Status_Error is propagated if the given file is not open. The exception Mode_Error is propagated if the mode of the given file is not Out_File or Append_File.

```
function Standard_Input return File_Type;
function Standard_Input return File_Access;
```

Returns the standard input file (see A.10), or an access value designating the standard input file, respectively.

```
function Standard_Output return File_Type;
function Standard_Output return File_Access;
```

Returns the standard output file (see A.10) or an access value designating the standard output file, respectively.

```
function Standard_Error return File_Type;
function Standard_Error return File_Access;
```

Returns the standard error file (see A.10), or an access value designating the standard error file, respectively.

The Form strings implicitly associated with the opening of Standard_Input, Standard_Output, and Standard_Error at the start of program execution are implementation defined. 13

```
function Current_Input return File_Type;
function Current_Input return File_Access;
```
14

Returns the current default input file, or an access value designating the current default input file, respectively. 15

```
function Current_Output return File_Type;
function Current_Output return File_Access;
```
16

Returns the current default output file, or an access value designating the current default output file, respectively. 17

```
function Current_Error return File_Type;
function Current_Error return File_Access;
```
18

Returns the current default error file, or an access value designating the current default error file, respectively. 19

```
procedure Flush (File : in File_Type);
procedure Flush;
```
20/1

The effect of Flush is the same as the corresponding subprogram in Sequential_IO (see A.8.2). If File is not explicitly specified, Current_Output is used. 21/4

Erroneous Execution

The execution of a program is erroneous if it invokes an operation on a current default input, default output, or default error file, and if the corresponding file object is closed or no longer exists. 22/1

This paragraph was deleted. 23/1

NOTE 1 The standard input, standard output, and standard error files cannot be opened, closed, reset, or deleted, because the parameter File of the corresponding procedures has the mode **in out**. 24

NOTE 2 The standard input, standard output, and standard error files are different file objects, but not necessarily different external files. 25

A.10.4 Specification of Line and Page Lengths

Static Semantics

The subprograms described in this subclause are concerned with the line and page structure of a file of mode Out_File or Append_File. They operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the current default output file. They provide for output of text with a specified maximum line length or page length. In these cases, line and page terminators are output implicitly and automatically when necessary. When line and page lengths are unbounded (that is, when they have the conventional value zero), as in the case of a newly opened file, new lines and new pages are only started when explicitly called for. 1/5

In all cases, the exception Status_Error is propagated if the file to be used is not open; the exception Mode_Error is propagated if the mode of the file is not Out_File or Append_File. 2

```
procedure Set_Line_Length(File : in File_Type; To : in Count);
procedure Set_Line_Length(To   : in Count);
```
3

Sets the maximum line length of the specified output or append file to the number of characters specified by To. The value zero for To specifies an unbounded line length. 4

The exception Use_Error is propagated if the specified line length is inappropriate for the associated external file.

```
procedure Set_Page_Length(File : in File_Type; To : in Count);
procedure Set_Page_Length(To   : in Count);
```

Sets the maximum page length of the specified output or append file to the number of lines specified by To. The value zero for To specifies an unbounded page length.

The exception Use_Error is propagated if the specified page length is inappropriate for the associated external file.

```
function Line_Length(File : in File_Type) return Count;
function Line_Length return Count;
```

Returns the maximum line length currently set for the specified output or append file, or zero if the line length is unbounded.

```
function Page_Length(File : in File_Type) return Count;
function Page_Length return Count;
```

Returns the maximum page length currently set for the specified output or append file, or zero if the page length is unbounded.

A.10.5 Operations on Columns, Lines, and Pages

Static Semantics

The subprograms described in this subclause provide for explicit control of line and page structure; they operate either on the file given as the first parameter, or, in the absence of such a file parameter, on the appropriate (input or output) current default file. The exception Status_Error is propagated by any of these subprograms if the file to be used is not open.

```
procedure New_Line(File : in File_Type; Spacing : in Positive_Count := 1);
procedure New_Line(Spacing : in Positive_Count := 1);
```

Operates on a file of mode Out_File or Append_File.

For a Spacing of one: Outputs a line terminator and sets the current column number to one. Then increments the current line number by one, except in the case that the current line number is already greater than or equal to the maximum page length, for a bounded page length; in that case a page terminator is output, the current page number is incremented by one, and the current line number is set to one.

For a Spacing greater than one, the above actions are performed Spacing times.

The exception Mode_Error is propagated if the mode is not Out_File or Append_File.

```
procedure Skip_Line(File  : in File_Type; Spacing : in Positive_Count := 1);
procedure Skip_Line(Spacing : in Positive_Count := 1);
```

Operates on a file of mode In_File.

For a Spacing of one: Reads and discards all characters until a line terminator has been read, and then sets the current column number to one. If the line terminator is not immediately followed by a page terminator, the current line number is incremented by one. Otherwise, if the line terminator is immediately followed by a page terminator, then the page terminator is skipped, the current page number is incremented by one, and the current line number is set to one.

For a Spacing greater than one, the above actions are performed Spacing times. 10

The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if an attempt is made to read a file terminator. 11

```
function End_Of_Line(File : in File_Type) return Boolean;
function End_Of_Line return Boolean;
```
12

Operates on a file of mode In_File. Returns True if a line terminator or a file terminator is next; otherwise, returns False. 13/3

The exception Mode_Error is propagated if the mode is not In_File. 14

```
procedure New_Page(File : in File_Type);
procedure New_Page;
```
15

Operates on a file of mode Out_File or Append_File. Outputs a line terminator if the current line is not terminated, or if the current page is empty (that is, if the current column and line numbers are both equal to one). Then outputs a page terminator, which terminates the current page. Adds one to the current page number and sets the current column and line numbers to one. 16

The exception Mode_Error is propagated if the mode is not Out_File or Append_File. 17

```
procedure Skip_Page(File : in File_Type);
procedure Skip_Page;
```
18

Operates on a file of mode In_File. Reads and discards all characters and line terminators until a page terminator has been read. Then adds one to the current page number, and sets the current column and line numbers to one. 19

The exception Mode_Error is propagated if the mode is not In_File. The exception End_Error is propagated if an attempt is made to read a file terminator. 20

```
function End_Of_Page(File : in File_Type) return Boolean;
function End_Of_Page return Boolean;
```
21

Operates on a file of mode In_File. Returns True if the combination of a line terminator and a page terminator is next, or if a file terminator is next; otherwise, returns False. 22/3

The exception Mode_Error is propagated if the mode is not In_File. 23

```
function End_Of_File(File : in File_Type) return Boolean;
function End_Of_File return Boolean;
```
24

Operates on a file of mode In_File. Returns True if a file terminator is next, or if the combination of a line, a page, and a file terminator is next; otherwise, returns False. 25/3

The exception Mode_Error is propagated if the mode is not In_File. 26

The following subprograms provide for the control of the current position of reading or writing in a file. In all cases, the default file is the current output file. 27

```
procedure Set_Col(File : in File_Type; To : in Positive_Count);
procedure Set_Col(To   : in Positive_Count);
```
28

If the file mode is Out_File or Append_File: 29

- If the value specified by To is greater than the current column number, outputs spaces, adding one to the current column number after each space, until the current column number equals the specified value. If the value specified by To is equal to the current column number, there is no effect. If the value specified by To is less than the current 30

column number, has the effect of calling New_Line (with a spacing of one), then outputs (To – 1) spaces, and sets the current column number to the specified value.

31 • The exception Layout_Error is propagated if the value specified by To exceeds Line_Length when the line length is bounded (that is, when it does not have the conventional value zero).

32 If the file mode is In_File:

33 • Reads (and discards) individual characters, line terminators, and page terminators, until the next character to be read has a column number that equals the value specified by To; there is no effect if the current column number already equals this value. Each transfer of a character or terminator maintains the current column, line, and page numbers in the same way as a Get procedure (see A.10.6). (Short lines will be skipped until a line is reached that has a character at the specified column position.)

34 • The exception End_Error is propagated if an attempt is made to read a file terminator.

35
```
procedure Set_Line(File : in File_Type; To : in Positive_Count);
procedure Set_Line(To   : in Positive_Count);
```

36 If the file mode is Out_File or Append_File:

37/3 • If the value specified by To is greater than the current line number, has the effect of repeatedly calling New_Line (with a spacing of one), until the current line number equals the specified value. If the value specified by To is equal to the current line number, there is no effect. If the value specified by To is less than the current line number, has the effect of calling New_Page followed, if To is greater than 1, by a call of New_Line with a spacing equal to (To – 1).

38 • The exception Layout_Error is propagated if the value specified by To exceeds Page_Length when the page length is bounded (that is, when it does not have the conventional value zero).

39 If the mode is In_File:

40 • Has the effect of repeatedly calling Skip_Line (with a spacing of one), until the current line number equals the value specified by To; there is no effect if the current line number already equals this value. (Short pages will be skipped until a page is reached that has a line at the specified line position.)

41 • The exception End_Error is propagated if an attempt is made to read a file terminator.

42
```
function Col(File : in File_Type) return Positive_Count;
function Col return Positive_Count;
```

43 Returns the current column number.

44 The exception Layout_Error is propagated if this number exceeds Count'Last.

45
```
function Line(File : in File_Type) return Positive_Count;
function Line return Positive_Count;
```

46 Returns the current line number.

47 The exception Layout_Error is propagated if this number exceeds Count'Last.

48
```
function Page(File : in File_Type) return Positive_Count;
function Page return Positive_Count;
```

49 Returns the current page number.

50 The exception Layout_Error is propagated if this number exceeds Count'Last.

The column number, line number, or page number are allowed to exceed Count'Last (as a consequence of the input or output of sufficiently many characters, lines, or pages). These events do not cause any exception to be propagated. However, a call of Col, Line, or Page propagates the exception Layout_Error if the corresponding number exceeds Count'Last. 51

NOTE A page terminator is always skipped whenever the preceding line terminator is skipped. An implementation can represent the combination of these terminators by a single character, provided that it is properly recognized on input. 52/5

A.10.6 Get and Put Procedures

Static Semantics

The procedures Get and Put for items of the type Character, String, numeric types, and enumeration types are described in subsequent subclauses. Features of these procedures that are common to most of these types are described in this subclause. The Get and Put procedures for items of type Character and String deal with individual character values; the Get and Put procedures for numeric and enumeration types treat the items as lexical elements. 1

All procedures Get and Put have forms with a file parameter, written first. Where this parameter is omitted, the appropriate (input or output) current default file is understood to be specified. Each procedure Get operates on a file of mode In_File. Each procedure Put operates on a file of mode Out_File or Append_File. 2

All procedures Get and Put maintain the current column, line, and page numbers of the specified file: the effect of each of these procedures upon these numbers is the result of the effects of individual transfers of characters and of individual output or skipping of terminators. Each transfer of a character adds one to the current column number. Each output of a line terminator sets the current column number to one and adds one to the current line number. Each output of a page terminator sets the current column and line numbers to one and adds one to the current page number. For input, each skipping of a line terminator sets the current column number to one and adds one to the current line number; each skipping of a page terminator sets the current column and line numbers to one and adds one to the current page number. Similar considerations apply to the procedures Get_Line, Put_Line, and Set_Col. 3

Several Get and Put procedures, for numeric and enumeration types, have *format* parameters which specify field lengths; these parameters are of the nonnegative subtype Field of the type Integer. 4

Input-output of enumeration values uses the syntax of the corresponding lexical elements. Any Get procedure for an enumeration type begins by skipping any leading blanks, or line or page terminators. A *blank* is defined as a space or a horizontal tabulation character. Next, characters are input only so long as the sequence input is an initial sequence of an identifier or of a character literal (in particular, input ceases when a line terminator is encountered). The character or line terminator that causes input to cease remains available for subsequent input. 5/2

For a numeric type, the Get procedures have a format parameter called Width. If the value given for this parameter is zero, the Get procedure proceeds in the same manner as for enumeration types, but using the syntax of numeric literals instead of that of enumeration literals. If a nonzero value is given, then exactly Width characters are input, or the characters up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. The syntax used for numeric literals is an extended syntax that allows a leading sign (but no intervening blanks, or line or page terminators) and that also allows (for real types) an integer literal as well as forms that have digits only before the point or only after the point. 6

Any Put procedure, for an item of a numeric or an enumeration type, outputs the value of the item as a numeric literal, identifier, or character literal, as appropriate. This is preceded by leading spaces if required 7/5

by the format parameters Width or Fore (as described in later subclauses), and then a minus sign for a negative value; for an enumeration type, the spaces follow instead of leading. The format given for a Put procedure is overridden if it is insufficiently wide, by using the minimum necessary width.

8 Two further cases arise for Put procedures for numeric and enumeration types, if the line length of the specified output file is bounded (that is, if it does not have the conventional value zero). If the number of characters to be output does not exceed the maximum line length, but is such that they cannot fit on the current line, starting from the current column, then (in effect) New_Line is called (with a spacing of one) before output of the item. Otherwise, if the number of characters exceeds the maximum line length, then the exception Layout_Error is propagated and nothing is output.

9 The exception Status_Error is propagated by any of the procedures Get, Get_Line, Put, and Put_Line if the file to be used is not open. The exception Mode_Error is propagated by the procedures Get and Get_Line if the mode of the file to be used is not In_File; and by the procedures Put and Put_Line, if the mode is not Out_File or Append_File.

10 The exception End_Error is propagated by a Get procedure if an attempt is made to skip a file terminator. The exception Data_Error is propagated by a Get procedure if the sequence finally input is not a lexical element corresponding to the type, in particular if no characters were input; for this test, leading blanks are ignored; for an item of a numeric type, when a sign is input, this rule applies to the succeeding numeric literal. The exception Layout_Error is propagated by a Put procedure that outputs to a parameter of type String, if the length of the actual string is insufficient for the output of the item.

Examples

11/5 In the examples, here and in A.10.8 and A.10.9, the string quotes and the lower case letter b are not transferred: they are shown only to reveal the layout and spaces.

12
```
N : Integer;
   ...
Get(N);
```

13
```
--  Characters at input   Sequence input    Value of N

--  bb–12535b             –12535            –12535
--  bb12_535e1b           12_535e1          125350
--  bb12_535e;            12_535e           (none) Data_Error raised
```

14 Example of overridden width parameter:

15
```
Put(Item => -23, Width => 2);  -- "–23"
```

A.10.7 Input-Output of Characters and Strings

Static Semantics

1 For an item of type Character the following procedures are provided:

2
```
procedure Get(File : in File_Type; Item : out Character);
procedure Get(Item : out Character);
```

3 After skipping any line terminators and any page terminators, reads the next character from the specified input file and returns the value of this character in the out parameter Item.

4 The exception End_Error is propagated if an attempt is made to skip a file terminator.

```
procedure Put(File : in File_Type; Item : in Character);
procedure Put(Item : in Character);
```

If the line length of the specified output file is bounded (that is, does not have the conventional value zero), and the current column number exceeds it, has the effect of calling New_Line with a spacing of one. Then, or otherwise, outputs the given character to the file.

```
procedure Look_Ahead (File        : in  File_Type;
                      Item        : out Character;
                      End_Of_Line : out Boolean);
procedure Look_Ahead (Item        : out Character;
                      End_Of_Line : out Boolean);
```

Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. Sets End_Of_Line to True if at end of line, including if at end of page or at end of file; in each of these cases the value of Item is not specified. Otherwise, End_Of_Line is set to False and Item is set to the next character (without consuming it) from the file.

```
procedure Get_Immediate(File : in  File_Type;
                        Item : out Character);
procedure Get_Immediate(Item : out Character);
```

Reads the next character, either control or graphic, from the specified File or the default input file. Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

```
procedure Get_Immediate(File      : in  File_Type;
                        Item      : out Character;
                        Available : out Boolean);
procedure Get_Immediate(Item      : out Character;
                        Available : out Boolean);
```

If a character, either control or graphic, is available from the specified File or the default input file, then the character is read; Available is True and Item contains the value of this character. If a character is not available, then Available is False and the value of Item is not specified. Status_Error is propagated if the file is not open. Mode_Error is propagated if the mode of the file is not In_File. End_Error is propagated if at the end of the file. The current column, line and page numbers for the file are not affected.

For an item of type String the following subprograms are provided:

```
procedure Get(File : in File_Type; Item : out String);
procedure Get(Item : out String);
```

Determines the length of the given string and attempts that number of Get operations for successive characters of the string (in particular, no operation is performed if the string is null).

```
procedure Put(File : in File_Type; Item : in String);
procedure Put(Item : in String);
```

Determines the length of the given string and attempts that number of Put operations for successive characters of the string (in particular, no operation is performed if the string is null).

```
function Get_Line(File : in File_Type) return String;
function Get_Line return String;
```

Returns a result string constructed by reading successive characters from the specified input file, and assigning them to successive characters of the result string. The result string has a lower bound of 1 and an upper bound of the number of characters read. Reading stops when the end of the line is met; Skip_Line is then (in effect) called with a spacing of 1.

17.3/2 Constraint_Error is raised if the length of the line exceeds Positive'Last; in this case, the line number and page number are unchanged, and the column number is unspecified but no less than it was before the call. The exception End_Error is propagated if an attempt is made to skip a file terminator.

18
```
procedure Get_Line(File : in File_Type;
                   Item : out String;
                   Last : out Natural);
procedure Get_Line(Item : out String;
                   Last : out Natural);
```

19 Reads successive characters from the specified input file and assigns them to successive characters of the specified string. Reading stops if the end of the string is met. Reading also stops if the end of the line is met before meeting the end of the string; in this case Skip_Line is (in effect) called with a spacing of 1. The values of characters not assigned are not specified.

20 If characters are read, returns in Last the index value such that Item(Last) is the last character assigned (the index of the first character assigned is Item'First). If no characters are read, returns in Last an index value that is one less than Item'First. The exception End_Error is propagated if an attempt is made to skip a file terminator.

21
```
procedure Put_Line(File : in File_Type; Item : in String);
procedure Put_Line(Item : in String);
```

22 Calls the procedure Put for the given string, and then the procedure New_Line with a spacing of one.

Implementation Advice

23 The Get_Immediate procedures should be implemented with unbuffered input. For a device such as a keyboard, input should be "available" if a key has already been typed, whereas for a disk file, input should always be available except at end of file. For a file associated with a keyboard-like device, any line-editing features of the underlying operating system should be disabled during the execution of Get_Immediate.

24 NOTE 1 Get_Immediate can be used to read a single key from the keyboard "immediately"; that is, without waiting for an end of line. In a call of Get_Immediate without the parameter Available, the caller will wait until a character is available.

25 NOTE 2 In a literal string parameter of Put, the enclosing string bracket characters are not output. Each doubled string bracket character in the enclosed string is output as a single string bracket character, as a consequence of the rule for string literals (see 2.6).

26 NOTE 3 A string read by Get or written by Put can extend over several lines. An implementation is allowed to assume that certain external files do not contain page terminators, in which case Get_Line and Skip_Line can return as soon as a line terminator is read.

A.10.8 Input-Output for Integer Types

Static Semantics

The following procedures are defined in the generic packages Integer_IO and Modular_IO, which have to be instantiated for the appropriate signed integer or modular type respectively (indicated by Num in the specifications). 1

Values are output as decimal or based literals, without low line characters or exponent, and, for Integer_IO, preceded by a minus sign if negative. The format (which includes any leading spaces and minus sign) can be specified by an optional field width parameter. Values of widths of fields in output formats are of the nonnegative integer subtype Field. Values of bases are of the integer subtype Number_Base. 2

```
subtype Number_Base is Integer range 2 .. 16;
```
3

The default field width and base to be used by output procedures are defined by the following variables that are declared in the generic packages Integer_IO and Modular_IO: 4

```
Default_Width : Field := Num'Width;
Default_Base  : Number_Base := 10;
```
5

The following procedures are provided: 6

```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```
7

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads a plus sign if present or (for a signed type only) a minus sign if present, then reads the longest possible sequence of characters matching the syntax of a numeric literal without a point. If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. 8

Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. 9

The exception Data_Error is propagated if the sequence of characters read does not form a legal integer literal or if the value obtained is not of the subtype Num. 10/3

```
procedure Put(File  : in File_Type;
              Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);

procedure Put(Item  : in Num;
              Width : in Field := Default_Width;
              Base  : in Number_Base := Default_Base);
```
11

Outputs the value of the parameter Item as an integer literal, with no low lines, no exponent, and no leading zeros (but a single zero for the value zero), and a preceding minus sign for a negative value. 12

If the resulting sequence of characters to be output has fewer than Width characters, then leading spaces are first output to make up the difference. 13

Uses the syntax for decimal literal if the parameter Base has the value ten (either explicitly or through Default_Base); otherwise, uses the syntax for based literal, with any letters in upper case. 14

15 `procedure Get(From : in String; Item : out Num; Last : out Positive);`

16 Reads an integer value from the beginning of the given string, following the same rules as the Get procedure that reads an integer value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

17 The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num.

18
```
procedure Put(To   : out String;
              Item : in Num;
              Base : in Number_Base := Default_Base);
```

19 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using the length of the given string as the value for Width.

20 Integer_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Integer_IO for the predefined type Integer:

21
```
with Ada.Text_IO;
package Ada.Integer_Text_IO is new Ada.Text_IO.Integer_IO(Integer);
```

22 For each predefined signed integer type, a nongeneric equivalent to Text_IO.Integer_IO is provided, with names such as Ada.Long_Integer_Text_IO.

Implementation Permissions

23/5 The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

24/3 *This paragraph was deleted.*

Examples

Examples of use of an instantiation of Text_IO.Integer_IO:

25/3
```
subtype Byte_Int is Integer range -127 .. 127;
package Int_IO is new Integer_IO(Byte_Int); use Int_IO;
-- default format used at instantiation,
-- Default_Width = 4, Default_Base = 10
```

26
```
Put(126);                              -- "b126"
Put(-126, 7);                          -- "bbb–126"
Put(126, Width => 13, Base => 2);      -- "bbb2#1111110#"
```

A.10.9 Input-Output for Real Types

Static Semantics

1 The following procedures are defined in the generic packages Float_IO, Fixed_IO, and Decimal_IO, which have to be instantiated for the appropriate floating point, ordinary fixed point, or decimal fixed point type respectively (indicated by Num in the specifications).

2 Values are output as decimal literals without low line characters. The format of each value output consists of a Fore field, a decimal point, an Aft field, and (if a nonzero Exp parameter is supplied) the letter E and an Exp field. The two possible formats thus correspond to:

3
```
Fore  .  Aft
```

4 and to:

5
```
Fore  .  Aft  E  Exp
```

without any spaces between these fields. The Fore field may include leading spaces, and a minus sign for negative values. The Aft field includes only decimal digits (possibly with trailing zeros). The Exp field includes the sign (plus or minus) and the exponent (possibly with leading zeros). 6

For floating point types, the default lengths of these fields are defined by the following variables that are declared in the generic package Float_IO: 7

```
Default_Fore : Field := 2;
Default_Aft  : Field := Num'Digits-1;
Default_Exp  : Field := 3;
```
8

For ordinary or decimal fixed point types, the default lengths of these fields are defined by the following variables that are declared in the generic packages Fixed_IO and Decimal_IO, respectively: 9

```
Default_Fore : Field := Num'Fore;
Default_Aft  : Field := Num'Aft;
Default_Exp  : Field := 0;
```
10

The following procedures are provided: 11

```
procedure Get(File : in File_Type; Item : out Num; Width : in Field := 0);
procedure Get(Item : out Num; Width : in Field := 0);
```
12

If the value of the parameter Width is zero, skips any leading blanks, line terminators, or page terminators, then reads the longest possible sequence of characters matching the syntax of any of the following (see 2.4): 13

- [+|–]numeric_literal 14
- [+|–]numeral.[exponent] 15
- [+|–].numeral[exponent] 16
- [+|–]base#based_numeral.#[exponent] 17
- [+|–]base#.based_numeral#[exponent] 18

If a nonzero value of Width is supplied, then exactly Width characters are input, or the characters (possibly none) up to a line terminator, whichever comes first; any skipped leading blanks are included in the count. 19

Returns in the parameter Item the value of type Num that corresponds to the sequence input, preserving the sign (positive if none has been specified) of a zero value if Num is a floating point type and Num'Signed_Zeros is True. 20

The exception Data_Error is propagated if the sequence input does not have the required syntax or if the value obtained is not of the subtype Num. 21

```
procedure Put(File : in File_Type;
              Item : in Num;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);

procedure Put(Item : in Num;
              Fore : in Field := Default_Fore;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```
22

Outputs the value of the parameter Item as a decimal literal with the format defined by Fore, Aft and Exp. If the value is negative, or if Num is a floating point type where Num'Signed_Zeros is True and the value is a negatively signed zero, then a minus sign is included in the integer part. If Exp has the value zero, then the integer part to be output has as many digits as are needed to 23

represent the integer part of the value of Item, overriding Fore if necessary, or consists of the digit zero if the value of Item has no integer part.

24 If Exp has a value greater than zero, then the integer part to be output has a single digit, which is nonzero except for the value 0.0 of Item.

25 In both cases, however, if the integer part to be output has fewer than Fore characters, including any minus sign, then leading spaces are first output to make up the difference. The number of digits of the fractional part is given by Aft, or is one if Aft equals zero. The value is rounded; a value of exactly one half in the last place is rounded away from zero.

26 If Exp has the value zero, there is no exponent part. If Exp has a value greater than zero, then the exponent part to be output has as many digits as are needed to represent the exponent part of the value of Item (for which a single digit integer part is used), and includes an initial sign (plus or minus). If the exponent part to be output has fewer than Exp characters, including the sign, then leading zeros precede the digits, to make up the difference. For the value 0.0 of Item, the exponent has the value zero.

27
```
procedure Get(From : in String; Item : out Num; Last : out Positive);
```

28 Reads a real value from the beginning of the given string, following the same rule as the Get procedure that reads a real value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Num that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

29 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the value obtained is not of the subtype Num.

30
```
procedure Put(To   : out String;
              Item : in Num;
              Aft  : in Field := Default_Aft;
              Exp  : in Field := Default_Exp);
```

31 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using a value for Fore such that the sequence of characters output exactly fills the string, including any leading spaces.

32 Float_Text_IO is a library package that is a nongeneric equivalent to Text_IO.Float_IO for the predefined type Float:

33
```
with Ada.Text_IO;
package Ada.Float_Text_IO is new Ada.Text_IO.Float_IO(Float);
```

34 For each predefined floating point type, a nongeneric equivalent to Text_IO.Float_IO is provided, with names such as Ada.Long_Float_Text_IO.

Implementation Permissions

35 An implementation may extend Get and Put for floating point types to support special values such as infinities and NaNs.

36/5 The implementation of Put may produce an output value with no greater accuracy than that which is supported for the base subtype. The additional accuracy, if any, of the value produced by Put when the number of requested digits in the integer and fractional parts exceeds the required accuracy is implementation defined.

The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required. 37/5

NOTE 1 For an item with a positive value, if output to a string exactly fills the string without leading spaces, then output of the corresponding negative value will propagate Layout_Error. 38

NOTE 2 The rules for the Value attribute (see 3.5) and the rules for Get are based on the same set of formats. 39

Examples

Examples of use of an instantiation of Text_IO.Float_IO: 40/5

```
package Real_IO is new Float_IO(Real); use Real_IO;                                    41
-- default format used at instantiation, Default_Exp = 3
X : Real := -123.4567;  -- digits 8   (see 3.5.7)                                      42
Put(X);   -- default format                          "–1.2345670E+02"                  43
Put(X, Fore => 5, Aft => 3, Exp => 2);   -- "bbb–1.235E+2"
Put(X, 5, 3, 0);                                     -- "b–123.457"
```

A.10.10 Input-Output for Enumeration Types

Static Semantics

The following procedures are defined in the generic package Enumeration_IO, which has to be instantiated for the appropriate enumeration type (indicated by Enum in the specification). 1

Values are output using either upper or lower case letters for identifiers. This is specified by the parameter Set, which is of the enumeration type Type_Set. 2

```
type Type_Set is (Lower_Case, Upper_Case);                                             3
```

The format (which includes any trailing spaces) can be specified by an optional field width parameter. The default field width and letter case are defined by the following variables that are declared in the generic package Enumeration_IO: 4

```
Default_Width   : Field := 0;                                                          5
Default_Setting : Type_Set := Upper_Case;
```

The following procedures are provided: 6

```
procedure Get(File : in File_Type; Item : out Enum);                                   7
procedure Get(Item : out Enum);
```

After skipping any leading blanks, line terminators, or page terminators, reads an identifier according to the syntax of this lexical element (lower and upper case being considered equivalent), or a character literal according to the syntax of this lexical element (including the apostrophes). Returns, in the parameter Item, the value of type Enum that corresponds to the sequence input. 8

The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype Enum. 9

10
```
procedure Put(File  : in File_Type;
              Item  : in Enum;
              Width : in Field := Default_Width;
              Set   : in Type_Set := Default_Setting);

procedure Put(Item  : in Enum;
              Width : in Field := Default_Width;
              Set   : in Type_Set := Default_Setting);
```

11 Outputs the value of the parameter Item as an enumeration literal (either an identifier or a character literal). The optional parameter Set indicates whether lower case or upper case is used for identifiers; it has no effect for character literals. If the sequence of characters produced has fewer than Width characters, then trailing spaces are finally output to make up the difference. If Enum is a character type, the sequence of characters produced is as for Enum'Image(Item), as modified by the Width and Set parameters.

12
```
procedure Get(From : in String; Item : out Enum; Last : out Positive);
```

13 Reads an enumeration value from the beginning of the given string, following the same rule as the Get procedure that reads an enumeration value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Enum that corresponds to the sequence input. Returns in Last the index value such that From(Last) is the last character read.

14 The exception Data_Error is propagated if the sequence input does not have the required syntax, or if the identifier or character literal does not correspond to a value of the subtype Enum.

15
```
procedure Put(To   : out String;
              Item : in Enum;
              Set  : in Type_Set := Default_Setting);
```

16 Outputs the value of the parameter Item to the given string, following the same rule as for output to a file, using the length of the given string as the value for Width.

17/1 Although the specification of the generic package Enumeration_IO would allow instantiation for an integer type, this is not the intended purpose of this generic package, and the effect of such instantiations is not defined by the language.

18 NOTE 1 There is a difference between Put defined for characters, and for enumeration values. Thus

19
```
Ada.Text_IO.Put('A');  -- outputs the character A
```

20
```
package Char_IO is new Ada.Text_IO.Enumeration_IO(Character);
Char_IO.Put('A');  -- outputs the character 'A', between apostrophes
```

21 NOTE 2 The type Boolean is an enumeration type, hence Enumeration_IO can be instantiated for this type.

A.10.11 Input-Output for Bounded Strings

1/2 The package Text_IO.Bounded_IO provides input-output in human-readable form for Bounded_Strings.

Static Semantics

2/2 The generic library package Text_IO.Bounded_IO has the following declaration:

3/5
```
with Ada.Strings.Bounded;
generic
   with package Bounded is
                      new Ada.Strings.Bounded.Generic_Bounded_Length (<>);
package Ada.Text_IO.Bounded_IO
   with Global => in out synchronized is
```

```
   procedure Put
      (File : in File_Type;
       Item : in Bounded.Bounded_String);                                    4/2

   procedure Put
      (Item : in Bounded.Bounded_String);                                    5/2

   procedure Put_Line
      (File : in File_Type;
       Item : in Bounded.Bounded_String);                                    6/2

   procedure Put_Line
      (Item : in Bounded.Bounded_String);                                    7/2

   function Get_Line
      (File : in File_Type)
      return Bounded.Bounded_String;                                         8/2

   function Get_Line
      return Bounded.Bounded_String;                                         9/2

   procedure Get_Line
      (File : in File_Type; Item : out Bounded.Bounded_String);              10/2

   procedure Get_Line
      (Item : out Bounded.Bounded_String);                                   11/2

end Ada.Text_IO.Bounded_IO;                                                  12/2
```

For an item of type Bounded_String, the following subprograms are provided: 13/2

```
procedure Put
   (File : in File_Type;
    Item : in Bounded.Bounded_String);                                       14/2
```

Equivalent to Text_IO.Put (File, Bounded.To_String(Item)); 15/2

```
procedure Put
   (Item : in Bounded.Bounded_String);                                       16/2
```

Equivalent to Text_IO.Put (Bounded.To_String(Item)); 17/2

```
procedure Put_Line
   (File : in File_Type;
    Item : in Bounded.Bounded_String);                                       18/2
```

Equivalent to Text_IO.Put_Line (File, Bounded.To_String(Item)); 19/2

```
procedure Put_Line
   (Item : in Bounded.Bounded_String);                                       20/2
```

Equivalent to Text_IO.Put_Line (Bounded.To_String(Item)); 21/2

```
function Get_Line
   (File : in File_Type)
   return Bounded.Bounded_String;                                            22/2
```

Returns Bounded.To_Bounded_String(Text_IO.Get_Line(File)); 23/2

```
function Get_Line
   return Bounded.Bounded_String;                                            24/2
```

Returns Bounded.To_Bounded_String(Text_IO.Get_Line); 25/2

```
procedure Get_Line
   (File : in File_Type; Item : out Bounded.Bounded_String);                 26/2
```

Equivalent to Item := Get_Line (File); 27/2

```
procedure Get_Line
   (Item : out Bounded.Bounded_String);                                      28/2
```

Equivalent to Item := Get_Line; 29/2

A.10.12 Input-Output for Unbounded Strings

1/2 The package Text_IO.Unbounded_IO provides input-output in human-readable form for Unbounded_Strings.

Static Semantics

2/2 The library package Text_IO.Unbounded_IO has the following declaration:

3/5
```
with Ada.Strings.Unbounded;
package Ada.Text_IO.Unbounded_IO
   with Global => in out synchronized is
```

4/2
```
   procedure Put
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);
```

5/2
```
   procedure Put
      (Item : in Strings.Unbounded.Unbounded_String);
```

6/2
```
   procedure Put_Line
      (File : in File_Type;
       Item : in Strings.Unbounded.Unbounded_String);
```

7/2
```
   procedure Put_Line
      (Item : in Strings.Unbounded.Unbounded_String);
```

8/2
```
   function Get_Line
      (File : in File_Type)
      return Strings.Unbounded.Unbounded_String;
```

9/2
```
   function Get_Line
      return Strings.Unbounded.Unbounded_String;
```

10/2
```
   procedure Get_Line
      (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```

11/2
```
   procedure Get_Line
      (Item : out Strings.Unbounded.Unbounded_String);
```

12/2
```
end Ada.Text_IO.Unbounded_IO;
```

13/2 For an item of type Unbounded_String, the following subprograms are provided:

14/2
```
procedure Put
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

15/2 Equivalent to Text_IO.Put (File, Strings.Unbounded.To_String(Item));

16/2
```
procedure Put
   (Item : in Strings.Unbounded.Unbounded_String);
```

17/2 Equivalent to Text_IO.Put (Strings.Unbounded.To_String(Item));

18/2
```
procedure Put_Line
   (File : in File_Type;
    Item : in Strings.Unbounded.Unbounded_String);
```

19/2 Equivalent to Text_IO.Put_Line (File, Strings.Unbounded.To_String(Item));

20/2
```
procedure Put_Line
   (Item : in Strings.Unbounded.Unbounded_String);
```

21/2 Equivalent to Text_IO.Put_Line (Strings.Unbounded.To_String(Item));

22/2
```
function Get_Line
   (File : in File_Type)
   return Strings.Unbounded.Unbounded_String;
```

23/2 Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line(File));

```
function Get_Line
   return Strings.Unbounded.Unbounded_String;
```
24/2

Returns Strings.Unbounded.To_Unbounded_String(Text_IO.Get_Line); 25/2

```
procedure Get_Line
   (File : in File_Type; Item : out Strings.Unbounded.Unbounded_String);
```
26/2

Equivalent to Item := Get_Line (File); 27/2

```
procedure Get_Line
   (Item : out Strings.Unbounded.Unbounded_String);
```
28/2

Equivalent to Item := Get_Line; 29/2

A.11 Wide Text Input-Output and Wide Wide Text Input-Output

The packages Wide_Text_IO and Wide_Wide_Text_IO provide facilities for input and output in human-readable form. Each file is read or written sequentially, as a sequence of wide characters (or wide wide characters) grouped into lines, and as a sequence of lines grouped into pages. 1/2

Static Semantics

The specification of package Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Character, and any occurrence of String is replaced by Wide_String. Nongeneric equivalents of Wide_Text_IO.Integer_IO and Wide_Text_IO.Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_Wide_Text_IO, Ada.Long_Integer_-Wide_Text_IO, Ada.Float_Wide_Text_IO, Ada.Long_Float_Wide_Text_IO. 2/2

The specification of package Wide_Wide_Text_IO is the same as that for Text_IO, except that in each Get, Look_Ahead, Get_Immediate, Get_Line, Put, and Put_Line subprogram, any occurrence of Character is replaced by Wide_Wide_Character, and any occurrence of String is replaced by Wide_Wide_String. Nongeneric equivalents of Wide_Wide_Text_IO.Integer_IO and Wide_Wide_Text_IO.Float_IO are provided (as for Text_IO) for each predefined numeric type, with names such as Ada.Integer_-Wide_Wide_Text_IO, Ada.Long_Integer_Wide_Wide_Text_IO, Ada.Float_Wide_Wide_Text_IO, Ada.Long_Float_Wide_Wide_Text_IO. 3/2

The specification of package Wide_Text_IO.Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Bounded_Wide_-String, and any occurrence of package Bounded is replaced by Wide_Bounded. The specification of package Wide_Wide_Text_IO.Wide_Wide_Bounded_IO is the same as that for Text_IO.Bounded_IO, except that any occurrence of Bounded_String is replaced by Bounded_Wide_Wide_String, and any occurrence of package Bounded is replaced by Wide_Wide_Bounded. 4/3

The specification of package Wide_Text_IO.Wide_Unbounded_IO is the same as that for Text_IO.-Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Unbounded_Wide_-String, and any occurrence of package Unbounded is replaced by Wide_Unbounded. The specification of package Wide_Wide_Text_IO.Wide_Wide_Unbounded_IO is the same as that for Text_IO.Unbounded_IO, except that any occurrence of Unbounded_String is replaced by Unbounded_-Wide_Wide_String, and any occurrence of package Unbounded is replaced by Wide_Wide_Unbounded. 5/3

A.12 Stream Input-Output

1/2 The packages Streams.Stream_IO, Text_IO.Text_Streams, Wide_Text_IO.Text_Streams, and Wide_Wide_Text_IO.Text_Streams provide stream-oriented operations on files.

A.12.1 The Package Streams.Stream_IO

1 The subprograms in the child package Streams.Stream_IO provide control over stream files. Access to a stream file is either sequential, via a call on Read or Write to transfer an array of stream elements, or positional (if supported by the implementation for the given file), by specifying a relative index for an element. Since a stream file can be converted to a Stream_Access value, calling stream-oriented attribute subprograms of different element types with the same Stream_Access value provides heterogeneous input-output. See 13.13 for a general discussion of streams.

Static Semantics

1.1/1 The elements of a stream file are stream elements. If positioning is supported for the specified external file, a current index and current size are maintained for the file as described in A.8. If positioning is not supported, a current index is not maintained, and the current size is implementation defined.

2 The library package Streams.Stream_IO has the following declaration:

3/5
```
with Ada.IO_Exceptions;
package Ada.Streams.Stream_IO
    with Preelaborate, Global => in out synchronized is

4       type Stream_Access is access all Root_Stream_Type'Class;

5/5     type File_Type is limited private
            with Preelaborable_Initialization;

6       type File_Mode is (In_File, Out_File, Append_File);

7       type    Count          is range 0 .. implementation-defined;
        subtype Positive_Count is Count range 1 .. Count'Last;
          -- Index into file, in stream elements.

8       procedure Create (File : in out File_Type;
                          Mode : in File_Mode := Out_File;
                          Name : in String    := "";
                          Form : in String    := "");

9       procedure Open (File : in out File_Type;
                        Mode : in File_Mode;
                        Name : in String;
                        Form : in String := "");

10      procedure Close  (File : in out File_Type);
        procedure Delete (File : in out File_Type);
        procedure Reset  (File : in out File_Type; Mode : in File_Mode);
        procedure Reset  (File : in out File_Type);

11      function Mode (File : in File_Type) return File_Mode;
        function Name (File : in File_Type) return String;
        function Form (File : in File_Type) return String;

12      function Is_Open     (File : in File_Type) return Boolean;
        function End_Of_File (File : in File_Type) return Boolean;

13      function Stream (File : in File_Type) return Stream_Access;
            -- Return stream access for use with T'Input and T'Output

14/1    This paragraph was deleted.
```

```
   -- Read array of stream elements from file                                    15/5
   procedure Read (File : in  File_Type;
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset;
                   From : in  Positive_Count)
      with Global => overriding in out File;

   procedure Read (File : in  File_Type;                                        16/5
                   Item : out Stream_Element_Array;
                   Last : out Stream_Element_Offset)
      with Global => overriding in out File;

This paragraph was deleted.                                                     17/1

   -- Write array of stream elements into file                                  18/5
   procedure Write (File : in File_Type;
                    Item : in Stream_Element_Array;
                    To   : in Positive_Count)
      with Global => overriding in out File;

   procedure Write (File : in File_Type;                                        19/5
                    Item : in Stream_Element_Array)
      with Global => overriding in out File;

This paragraph was deleted.                                                     20/1

   -- Operations on position within file                                        21

   procedure Set_Index(File : in File_Type; To : in Positive_Count)             22/5
      with Global => overriding in out File;

   function Index(File : in File_Type) return Positive_Count;                   23
   function Size (File : in File_Type) return Count;

   procedure Set_Mode(File : in out File_Type; Mode : in File_Mode);            24

   procedure Flush(File : in File_Type);                                        25/1

   -- exceptions                                                                26
   Status_Error : exception renames IO_Exceptions.Status_Error;
   Mode_Error   : exception renames IO_Exceptions.Mode_Error;
   Name_Error   : exception renames IO_Exceptions.Name_Error;
   Use_Error    : exception renames IO_Exceptions.Use_Error;
   Device_Error : exception renames IO_Exceptions.Device_Error;
   End_Error    : exception renames IO_Exceptions.End_Error;
   Data_Error   : exception renames IO_Exceptions.Data_Error;

package Wide_File_Names is                                                      26.1/5

   -- File management                                                           26.2/5

   procedure Create (File : in out File_Type;                                   26.3/5
                     Mode : in File_Mode := Out_File;
                     Name : in Wide_String := "";
                     Form : in Wide_String := "");

   procedure Open (File : in out File_Type;                                     26.4/5
                   Mode : in File_Mode;
                   Name : in Wide_String;
                   Form : in Wide_String := "");

   function Name (File : in File_Type) return Wide_String;                      26.5/5

   function Form (File : in File_Type) return Wide_String;                      26.6/5

end Wide_File_Names;                                                            26.7/5

package Wide_Wide_File_Names is                                                 26.8/5

   -- File management                                                           26.9/5

   procedure Create (File : in out File_Type;                                   26.10/5
                     Mode : in File_Mode := Out_File;
                     Name : in Wide_Wide_String := "";
                     Form : in Wide_Wide_String := "");
```

26.11/5
```
        procedure Open (File : in out File_Type;
                        Mode : in File_Mode;
                        Name : in Wide_Wide_String;
                        Form : in Wide_Wide_String := "");
```
26.12/5
```
        function Name (File : in File_Type) return Wide_Wide_String;
```
26.13/5
```
        function Form (File : in File_Type) return Wide_Wide_String;
```
26.14/5
```
    end Wide_Wide_File_Names;
```
27
```
private
   ... -- not specified by the language
end Ada.Streams.Stream_IO;
```

27.1/2 The type File_Type needs finalization (see 7.6).

28/5 The subprograms given in A.8.2 for the control of external files (Create, Open, Close, Delete, Reset, Mode, Name, Form, Is_Open, and Flush) are available for stream files.

28.1/2 The End_Of_File function:

28.2/2 • Propagates Mode_Error if the mode of the file is not In_File;

28.3/3 • If positioning is supported for the given external file, the function returns True if the current index exceeds the size of the external file; otherwise, it returns False;

28.4/3 • If positioning is not supported for the given external file, the function returns True if no more elements can be read from the given file; otherwise, it returns False.

28.5/2 The Set_Mode procedure sets the mode of the file. If the new mode is Append_File, the file is positioned to its end; otherwise, the position in the file is unchanged.

28.6/4 *This paragraph was deleted.*

29/1 The Stream function returns a Stream_Access result from a File_Type object, thus allowing the stream-oriented attributes Read, Write, Input, and Output to be used on the same file for multiple types. Stream propagates Status_Error if File is not open.

30/2 The procedures Read and Write are equivalent to the corresponding operations in the package Streams. Read propagates Mode_Error if the mode of File is not In_File. Write propagates Mode_Error if the mode of File is not Out_File or Append_File. The Read procedure with a Positive_Count parameter starts reading at the specified index. The Write procedure with a Positive_Count parameter starts writing at the specified index. For a file that supports positioning, Read without a Positive_Count parameter starts reading at the current index, and Write without a Positive_Count parameter starts writing at the current index.

30.1/1 The Size function returns the current size of the file.

31/1 The Index function returns the current index.

32 The Set_Index procedure sets the current index to the specified value.

32.1/1 If positioning is supported for the external file, the current index is maintained as follows:

32.2/1 • For Open and Create, if the Mode parameter is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one.

32.3/1 • For Reset, if the Mode parameter is Append_File, or no Mode parameter is given and the current mode is Append_File, the current index is set to the current size of the file plus one; otherwise, the current index is set to one.

32.4/1 • For Set_Mode, if the new mode is Append_File, the current index is set to current size plus one; otherwise, the current index is unchanged.

- For Read and Write without a Positive_Count parameter, the current index is incremented by the number of stream elements read or written. 32.5/1
- For Read and Write with a Positive_Count parameter, the value of the current index is set to the value of the Positive_Count parameter plus the number of stream elements read or written. 32.6/1

If positioning is not supported for the given file, then a call of Index or Set_Index propagates Use_Error. Similarly, a call of Read or Write with a Positive_Count parameter propagates Use_Error. 33

Paragraphs 34 through 36 were deleted.

Erroneous Execution

If the File_Type object passed to the Stream function is later closed or finalized, and the stream-oriented attributes are subsequently called (explicitly or implicitly) on the Stream_Access value returned by Stream, execution is erroneous. This rule applies even if the File_Type object was opened again after it had been closed. 36.1/1

A.12.2 The Package Text_IO.Text_Streams

The package Text_IO.Text_Streams provides a function for treating a text file as a stream. 1

Static Semantics

The library package Text_IO.Text_Streams has the following declaration: 2

```
with Ada.Streams;                                                          3/5
package Ada.Text_IO.Text_Streams
   with Global => in out synchronized is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;

   function Stream (File : in File_Type) return Stream_Access;             4
end Ada.Text_IO.Text_Streams;
```

The Stream function has the same effect as the corresponding function in Streams.Stream_IO. 5

NOTE 1 The ability to obtain a stream for a text file allows Current_Input, Current_Output, and Current_Error to be processed with the functionality of streams, including the mixing of text and binary input-output, and the mixing of binary input-output for different types. 6

NOTE 2 Performing operations on the stream associated with a text file does not affect the column, line, or page counts. 7

A.12.3 The Package Wide_Text_IO.Text_Streams

The package Wide_Text_IO.Text_Streams provides a function for treating a wide text file as a stream. 1

Static Semantics

The library package Wide_Text_IO.Text_Streams has the following declaration: 2

```
with Ada.Streams;                                                          3/5
package Ada.Wide_Text_IO.Text_Streams
   with Global => in out synchronized is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;

   function Stream (File : in File_Type) return Stream_Access;             4
end Ada.Wide_Text_IO.Text_Streams;
```

The Stream function has the same effect as the corresponding function in Streams.Stream_IO. 5

A.12.4 The Package Wide_Wide_Text_IO.Text_Streams

1/2 The package Wide_Wide_Text_IO.Text_Streams provides a function for treating a wide wide text file as a stream.

Static Semantics

2/2 The library package Wide_Wide_Text_IO.Text_Streams has the following declaration:

3/5
```
with Ada.Streams;
package Ada.Wide_Wide_Text_IO.Text_Streams
   with Global => in out synchronized is
   type Stream_Access is access all Streams.Root_Stream_Type'Class;
```

4/2
```
   function Stream (File : in File_Type) return Stream_Access;
end Ada.Wide_Wide_Text_IO.Text_Streams;
```

5/2 The Stream function has the same effect as the corresponding function in Streams.Stream_IO.

A.13 Exceptions in Input-Output

1/5 The package IO_Exceptions defines the exceptions used by the predefined input-output packages.

Static Semantics

2 The library package IO_Exceptions has the following declaration:

3/5
```
package Ada.IO_Exceptions
   with Pure is
```

4
```
   Status_Error : exception;
   Mode_Error   : exception;
   Name_Error   : exception;
   Use_Error    : exception;
   Device_Error : exception;
   End_Error    : exception;
   Data_Error   : exception;
   Layout_Error : exception;
```

5
```
end Ada.IO_Exceptions;
```

6 If more than one error condition exists, the corresponding exception that appears earliest in the following list is the one that is propagated.

7 The exception Status_Error is propagated by an attempt to operate upon a file that is not open, and by an attempt to open a file that is already open.

8 The exception Mode_Error is propagated by an attempt to read from, or test for the end of, a file whose current mode is Out_File or Append_File, and also by an attempt to write to a file whose current mode is In_File. In the case of Text_IO, the exception Mode_Error is also propagated by specifying a file whose current mode is Out_File or Append_File in a call of Set_Input, Skip_Line, End_Of_Line, Skip_Page, or End_Of_Page; and by specifying a file whose current mode is In_File in a call of Set_Output, Set_Line_Length, Set_Page_Length, Line_Length, Page_Length, New_Line, or New_Page.

9 The exception Name_Error is propagated by a call of Create or Open if the string given for the parameter Name does not allow the identification of an external file. For example, this exception is propagated if the string is improper, or, alternatively, if either none or more than one external file corresponds to the string.

10 The exception Use_Error is propagated if an operation is attempted that is not possible for reasons that depend on characteristics of the external file. For example, this exception is propagated by the procedure

Create, among other circumstances, if the given mode is Out_File but the form specifies an input only device, if the parameter Form specifies invalid access rights, or if an external file with the given name already exists and overwriting is not allowed.

The exception Device_Error is propagated if an input-output operation cannot be completed because of a malfunction of the underlying system. 11

The exception End_Error is propagated by an attempt to skip (read past) the end of a file. 12

The exception Data_Error can be propagated by the procedure Read (or by the Read attribute) if the element read cannot be interpreted as a value of the required subtype. This exception is also propagated by a procedure Get (defined in the package Text_IO) if the input character sequence fails to satisfy the required syntax, or if the value input does not belong to the range of the required subtype. 13

The exception Layout_Error is propagated (in text input-output) by Col, Line, or Page if the value returned exceeds Count'Last. The exception Layout_Error is also propagated on output by an attempt to set column or line numbers in excess of specified maximum line or page lengths, respectively (excluding the unbounded cases). It is also propagated by an attempt to Put too many characters to a string. 14

These exceptions are also propagated by various other language-defined packages and operations, see the definition of those entities for other reasons that these exceptions are propagated. 14.1/3

Documentation Requirements

The implementation shall document the conditions under which Name_Error, Use_Error and Device_Error are propagated. 15

Implementation Permissions

When the associated check is complex, it is optional to propagate Data_Error as part of a procedure Read (or the Read attribute) when the value read cannot be interpreted as a value of the required subtype. 16/5

Erroneous Execution

If the element read by the procedure Read (or by the Read attribute) cannot be interpreted as a value of the required subtype, but this is not detected and Data_Error is not propagated, then the resulting value can be abnormal, and subsequent references to the value can lead to erroneous execution, as explained in 13.9.1. 17

A.14 File Sharing

Dynamic Semantics

It is not specified by the language whether the same external file can be associated with more than one file object. If such sharing is supported by the implementation, the following effects are defined: 1

- Operations on one text file object do not affect the column, line, and page numbers of any other file object. 2
- *This paragraph was deleted.* 3/1
- For direct and stream files, the current index is a property of each file object; an operation on one file object does not affect the current index of any other file object. 4
- For direct and stream files, the current size of the file is a property of the external file. 5

All other effects are identical. 6

A.15 The Package Command_Line

1 The package Command_Line allows a program to obtain the values of its arguments and to set the exit status code to be returned on normal termination.

Static Semantics

2 The library package Ada.Command_Line has the following declaration:

3/5
```
package Ada.Command_Line
  with Preelaborate, Nonblocking, Global => in out synchronized is
```

4
```
  function Argument_Count return Natural;
```

5
```
  function Argument (Number : in Positive) return String;
```

6
```
  function Command_Name return String;
```

7
```
  type Exit_Status is implementation-defined integer type;
```

8
```
  Success : constant Exit_Status;
  Failure : constant Exit_Status;
```

9
```
  procedure Set_Exit_Status (Code : in Exit_Status);
```

10
```
private
  ... -- not specified by the language
end Ada.Command_Line;
```

11
```
function Argument_Count return Natural;
```

12/3 If the external execution environment supports passing arguments to a program, then Argument_Count returns the number of arguments passed to the program invoking the function. Otherwise, it returns 0. The meaning of "number of arguments" is implementation defined.

13
```
function Argument (Number : in Positive) return String;
```

14/5 If the external execution environment supports passing arguments to a program, then Argument returns an implementation-defined value with lower bound 1 corresponding to the argument at relative position Number. If Number is outside the range 1..Argument_Count, then Constraint_Error is propagated.

15
```
function Command_Name return String;
```

16/5 If the external execution environment supports passing arguments to a program, then Command_Name returns an implementation-defined value with lower bound 1 corresponding to the name of the command invoking the program; otherwise, Command_Name returns the null string.

16.1/1
```
type Exit_Status is implementation-defined integer type;
```

17 The type Exit_Status represents the range of exit status values supported by the external execution environment. The constants Success and Failure correspond to success and failure, respectively.

18
```
procedure Set_Exit_Status (Code : in Exit_Status);
```

19 If the external execution environment supports returning an exit status from a program, then Set_Exit_Status sets Code as the status. Normal termination of a program returns as the exit status the value most recently set by Set_Exit_Status, or, if no such value has been set, then the value Success. If a program terminates abnormally, the status set by Set_Exit_Status is ignored, and an implementation-defined exit status value is set.

If the external execution environment does not support returning an exit value from a program, then Set_Exit_Status does nothing. 20

Implementation Permissions

An alternative declaration is allowed for package Command_Line if different functionality is appropriate for the external execution environment. 21

NOTE Argument_Count, Argument, and Command_Name correspond to the C language's argc, argv[n] (for n>0) and argv[0], respectively. 22

A.15.1 The Packages Wide_Command_Line and Wide_Wide_Command_Line

The packages Wide_Command_Line and Wide_Wide_Command_Line allow a program to obtain the values of its arguments and to set the exit status code to be returned on normal termination. 1/5

Static Semantics

The specification of package Wide_Command_Line is the same as for Command_Line, except that each occurrence of String is replaced by Wide_String. 2/5

The specification of package Wide_Wide_Command_Line is the same as for Command_Line, except that each occurrence of String is replaced by Wide_Wide_String. 3/5

A.16 The Package Directories

The package Directories provides operations for manipulating files and directories, and their names. 1/2

Static Semantics

The library package Directories has the following declaration: 2/2

```
with Ada.IO_Exceptions;                                                    3/5
with Ada.Calendar;
package Ada.Directories
   with Global => in out synchronized is
   -- Directory and file operations:                                       4/2
   function Current_Directory return String;                               5/2
   procedure Set_Directory (Directory : in String);                        6/2
   procedure Create_Directory (New_Directory : in String;                  7/2
                               Form          : in String := "");
   procedure Delete_Directory (Directory : in String);                     8/2
   procedure Create_Path (New_Directory : in String;                       9/2
                          Form          : in String := "");
   procedure Delete_Tree (Directory : in String);                          10/2
   procedure Delete_File (Name : in String);                               11/2
   procedure Rename (Old_Name, New_Name : in String);                      12/2
   procedure Copy_File (Source_Name,                                       13/2
                        Target_Name : in String;
                        Form        : in String := "");
   -- File and directory name operations:                                  14/2
   function Full_Name (Name : in String) return String                     15/5
      with Nonblocking;
```

```
16/5        function Simple_Name (Name : in String) return String
               with Nonblocking;
17/5        function Containing_Directory (Name : in String) return String
               with Nonblocking;
18/5        function Extension (Name : in String) return String
               with Nonblocking;
19/5        function Base_Name (Name : in String) return String
               with Nonblocking;
20/5        function Compose (Containing_Directory : in String := "";
                              Name                 : in String;
                              Extension            : in String := "") return String
               with Nonblocking;
20.1/3      type Name_Case_Kind is
               (Unknown, Case_Sensitive, Case_Insensitive, Case_Preserving);
20.2/3      function Name_Case_Equivalence (Name : in String) return Name_Case_Kind;
21/2        -- File and directory queries:
22/2        type File_Kind is (Directory, Ordinary_File, Special_File);
23/2        type File_Size is range 0 .. implementation-defined;
24/2        function Exists (Name : in String) return Boolean;
25/2        function Kind (Name : in String) return File_Kind;
26/2        function Size (Name : in String) return File_Size;
27/2        function Modification_Time (Name : in String) return Ada.Calendar.Time;
28/2        -- Directory searching:
29/2        type Directory_Entry_Type is limited private;
30/2        type Filter_Type is array (File_Kind) of Boolean;
31/2        type Search_Type is limited private;
32/2        procedure Start_Search (Search    : in out Search_Type;
                                    Directory : in String;
                                    Pattern   : in String;
                                    Filter    : in Filter_Type := (others => True));
33/2        procedure End_Search (Search : in out Search_Type);
34/2        function More_Entries (Search : in Search_Type) return Boolean;
35/2        procedure Get_Next_Entry (Search : in out Search_Type;
                                      Directory_Entry : out Directory_Entry_Type);
36/5        procedure Search (
               Directory : in String;
               Pattern   : in String;
               Filter    : in Filter_Type := (others => True);
               Process   : not null access procedure (
                   Directory_Entry : in Directory_Entry_Type))
               with Allows_Exit;
37/2        -- Operations on Directory Entries:
38/2        function Simple_Name (Directory_Entry : in Directory_Entry_Type)
                return String;
39/2        function Full_Name (Directory_Entry : in Directory_Entry_Type)
                return String;
40/2        function Kind (Directory_Entry : in Directory_Entry_Type)
                return File_Kind;
41/2        function Size (Directory_Entry : in Directory_Entry_Type)
                return File_Size;
42/2        function Modification_Time (Directory_Entry : in Directory_Entry_Type)
                return Ada.Calendar.Time;
```

```
   Status_Error : exception renames Ada.IO_Exceptions.Status_Error;
   Name_Error   : exception renames Ada.IO_Exceptions.Name_Error;
   Use_Error    : exception renames Ada.IO_Exceptions.Use_Error;
   Device_Error : exception renames Ada.IO_Exceptions.Device_Error;
```
43/2

```
private
    ... -- not specified by the language
end Ada.Directories;
```
44/3

External files may be classified as directories, special files, or ordinary files. A *directory* is an external file that is a container for files on the target system. A *special file* is an external file that cannot be created or read by a predefined Ada input-output package. External files that are not special files or directories are called *ordinary files*. 45/2

A *file name* is a string identifying an external file. Similarly, a *directory name* is a string identifying a directory. The interpretation of file names and directory names is implementation defined. 46/2

The *full name* of an external file is a full specification of the name of the file. If the external environment allows alternative specifications of the name (for example, abbreviations), the full name should not use such alternatives. A full name typically will include the names of all of the directories that contain the item. The *simple name* of an external file is the name of the item, not including any containing directory names. Unless otherwise specified, a file name or directory name parameter in a call to a predefined Ada input-output subprogram can be a full name, a simple name, or any other form of name supported by the implementation. 47/2

A *root directory* is a directory that has no containing directory. 47.1/5

The *default directory* is the directory that is used if a directory or file name is not a full name (that is, when the name does not fully identify all of the containing directories). 48/2

A *directory entry* is a single item in a directory, identifying a single external file (including directories and special files). 49/2

For each function that returns a string, the lower bound of the returned value is 1. 50/2

The following file and directory operations are provided: 51/2

```
function Current_Directory return String;
```
52/2

Returns the full directory name for the current default directory. The name returned shall be suitable for a future call to Set_Directory. The exception Use_Error is propagated if a default directory is not supported by the external environment. 53/2

```
procedure Set_Directory (Directory : in String);
```
54/2

Sets the current default directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support making Directory (in the absence of Name_Error) a default directory. 55/2

```
procedure Create_Directory (New_Directory : in String;
                            Form          : in String := "");
```
56/2

Creates a directory with name New_Directory. The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of a directory. The exception 57/2

Use_Error is propagated if the external environment does not support the creation of a directory with the given name (in the absence of Name_Error) and form.

58/2 `procedure Delete_Directory (Directory : in String);`

59/3 Deletes an existing empty directory with name Directory. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the directory is not empty or the external environment does not support the deletion of the directory with the given name (in the absence of Name_Error).

60/2
```
procedure Create_Path (New_Directory : in String;
                       Form          : in String := "");
```

61/3 Creates zero or more directories with name New_Directory. Each nonexistent directory named by New_Directory is created. For example, on a typical Unix system, Create_Path ("/usr/me/my"); would create directory "me" in directory "usr", then create directory "my" in directory "me". The Form parameter can be used to give system-dependent characteristics of the directory; the interpretation of the Form parameter is implementation defined. A null string for Form specifies the use of the default options of the implementation of the new directory. The exception Name_Error is propagated if the string given as New_Directory does not allow the identification of any directory. The exception Use_Error is propagated if the external environment does not support the creation of any directories with the given name (in the absence of Name_Error) and form. If Use_Error is propagated, it is unspecified whether a portion of the directory path is created.

62/2 `procedure Delete_Tree (Directory : in String);`

63/2 Deletes an existing directory with name Directory. The directory and all of its contents (possibly including other directories) are deleted. The exception Name_Error is propagated if the string given as Directory does not identify an existing directory. The exception Use_Error is propagated if the external environment does not support the deletion of the directory or some portion of its contents with the given name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified whether a portion of the contents of the directory is deleted.

64/2 `procedure Delete_File (Name : in String);`

65/2 Deletes an existing ordinary or special file with name Name. The exception Name_Error is propagated if the string given as Name does not identify an existing ordinary or special external file. The exception Use_Error is propagated if the external environment does not support the deletion of the file with the given name (in the absence of Name_Error).

66/2 `procedure Rename (Old_Name, New_Name : in String);`

67/3 Renames an existing external file (including directories) with name Old_Name to New_Name. The exception Name_Error is propagated if the string given as Old_Name does not identify an existing external file or if the string given as New_Name does not allow the identification of an external file. The exception Use_Error is propagated if the external environment does not support the renaming of the file with the given name (in the absence of Name_Error). In particular, Use_Error is propagated if a file or directory already exists with name New_Name.

68/3
```
procedure Copy_File (Source_Name,
                     Target_Name : in String;
                     Form        : in String := "");
```

69/3 Copies the contents of the existing external file with name Source_Name to an external file with name Target_Name. The resulting external file is a duplicate of the source external file. The

Form parameter can be used to give system-dependent characteristics of the resulting external file; the interpretation of the Form parameter is implementation defined. Exception Name_Error is propagated if the string given as Source_Name does not identify an existing external ordinary or special file, or if the string given as Target_Name does not allow the identification of an external file. The exception Use_Error is propagated if the external environment does not support creating the file with the name given by Target_Name and form given by Form, or copying of the file with the name given by Source_Name (in the absence of Name_Error). If Use_Error is propagated, it is unspecified whether a portion of the file is copied.

The following file and directory name operations are provided: 70/2

```
function Full_Name (Name : in String) return String;
```
71/2

Returns the full name corresponding to the file name specified by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files). 72/2

```
function Simple_Name (Name : in String) return String;
```
73/2

Returns the simple name portion of the file name specified by Name. The simple name of a root directory is a name of the root itself. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files). 74/5

```
function Containing_Directory (Name : in String) return String;
```
75/2

Returns the name of the containing directory of the external file (including directories) identified by Name. (If more than one directory can contain Name, the directory name returned is implementation defined.) The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. The exception Use_Error is propagated if the external file does not have a containing directory. 76/2

```
function Extension (Name : in String) return String;
```
77/2

Returns the extension name corresponding to Name. The extension name is a portion of a simple name (not including any separator characters), typically used to identify the file class. If the external environment does not have extension names, then the null string is returned. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file. 78/2

```
function Base_Name (Name : in String) return String;
```
79/2

Returns the base name corresponding to Name. The base name is the remainder of a simple name after removing any extension and extension separators. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files). 80/2

```
function Compose (Containing_Directory : in String := "";
                  Name                 : in String;
                  Extension            : in String := "") return String;
```
81/2

Returns the name of the external file with the specified Containing_Directory, Name, and Extension. If Extension is the null string, then Name is interpreted as a simple name; otherwise, Name is interpreted as a base name. The exception Name_Error is propagated if: 82/5

- the string given as Containing_Directory is not null and does not allow the identification of a directory; 82.1/5

82.2/5 • the string given as Extension is not null and is not a possible extension;

82.3/5 • the string given as Name is not a possible simple name (if Extension is null) or base name (if Extension is nonnull); or

82.4/5 • the string given as Name is a root directory, and Containing_Directory or Extension is nonnull.

82.5/3 `function Name_Case_Equivalence (Name : in String) return Name_Case_Kind;`

82.6/3 Returns the file name equivalence rule for the directory containing Name. Raises Name_Error if Name is not a full name. Returns Case_Sensitive if file names that differ only in the case of letters are considered different names. If file names that differ only in the case of letters are considered the same name, then Case_Preserving is returned if names have the case of the file name used when a file is created; and Case_Insensitive is returned otherwise. Returns Unknown if the file name equivalence is not known.

83/2 The following file and directory queries and types are provided:

84/2 `type File_Kind is (Directory, Ordinary_File, Special_File);`

85/2 The type File_Kind represents the kind of file represented by an external file or directory.

86/2 `type File_Size is range 0 .. implementation-defined;`

87/2 The type File_Size represents the size of an external file.

88/2 `function Exists (Name : in String) return Boolean;`

89/2 Returns True if an external file represented by Name exists, and False otherwise. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

90/2 `function Kind (Name : in String) return File_Kind;`

91/2 Returns the kind of external file represented by Name. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file.

92/2 `function Size (Name : in String) return File_Size;`

93/2 Returns the size of the external file represented by Name. The size of an external file is the number of stream elements contained in the file. If the external file is not an ordinary file, the result is implementation defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

94/2 `function Modification_Time (Name : in String) return Ada.Calendar.Time;`

95/2 Returns the time that the external file represented by Name was most recently modified. If the external file is not an ordinary file, the result is implementation defined. The exception Name_Error is propagated if the string given as Name does not allow the identification of an existing external file. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file with the name given by Name (in the absence of Name_Error).

96/2 The following directory searching operations and types are provided:

97/2 `type Directory_Entry_Type is limited private;`

98/2 The type Directory_Entry_Type represents a single item in a directory. These items can only be created by the Get_Next_Entry procedure in this package. Information about the item can be

obtained from the functions declared in this package. A default-initialized object of this type is invalid; objects returned from Get_Next_Entry are valid.

```
type Filter_Type is array (File_Kind) of Boolean;
```
99/2

The type Filter_Type specifies which directory entries are provided from a search operation. If the Directory component is True, directory entries representing directories are provided. If the Ordinary_File component is True, directory entries representing ordinary files are provided. If the Special_File component is True, directory entries representing special files are provided.
100/2

```
type Search_Type is limited private;
```
101/2

The type Search_Type contains the state of a directory search. A default-initialized Search_Type object has no entries available (function More_Entries returns False). Type Search_Type needs finalization (see 7.6).
102/2

```
procedure Start_Search (Search    : in out Search_Type;
                        Directory : in String;
                        Pattern   : in String;
                        Filter    : in Filter_Type := (others => True));
```
103/2

Starts a search in the directory named by Directory for entries matching Pattern and Filter. Pattern represents a pattern for matching file names. If Pattern is the null string, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation defined. Only items that match Filter will be returned. After a successful call on Start_Search, the object Search may have entries available, but it may have no entries available if no files or directories match Pattern and Filter. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error). When Start_Search propagates Name_Error or Use_Error, the object Search will have no entries available.
104/3

```
procedure End_Search (Search : in out Search_Type);
```
105/2

Ends the search represented by Search. After a successful call on End_Search, the object Search will have no entries available.
106/2

```
function More_Entries (Search : in Search_Type) return Boolean;
```
107/2

Returns True if more entries are available to be returned by a call to Get_Next_Entry for the specified search object, and False otherwise.
108/2

```
procedure Get_Next_Entry (Search : in out Search_Type;
                          Directory_Entry : out Directory_Entry_Type);
```
109/2

Returns the next Directory_Entry for the search described by Search that matches the pattern and filter. If no further matches are available, Status_Error is raised. It is implementation defined as to whether the results returned by this subprogram are altered if the contents of the directory are altered while the Search object is valid (for example, by another program). The exception Use_Error is propagated if the external environment does not support continued searching of the directory represented by Search.
110/3

111/5
```
procedure Search (
   Directory : in String;
   Pattern   : in String;
   Filter    : in Filter_Type := (others => True);
   Process   : not null access procedure (
       Directory_Entry : in Directory_Entry_Type))
   with Allows_Exit;
```

112/3 Searches in the directory named by Directory for entries matching Pattern and Filter. The subprogram designated by Process is called with each matching entry in turn. Pattern represents a pattern for matching file names. If Pattern is the null string, all items in the directory are matched; otherwise, the interpretation of Pattern is implementation defined. Only items that match Filter will be returned. The exception Name_Error is propagated if the string given by Directory does not identify an existing directory, or if Pattern does not allow the identification of any possible external file or directory. The exception Use_Error is propagated if the external environment does not support the searching of the directory with the given name (in the absence of Name_Error).

113/2
```
function Simple_Name (Directory_Entry : in Directory_Entry_Type)
     return String;
```

114/2 Returns the simple external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid.

115/2
```
function Full_Name (Directory_Entry : in Directory_Entry_Type)
     return String;
```

116/2 Returns the full external name of the external file (including directories) represented by Directory_Entry. The format of the name returned is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid.

117/2
```
function Kind (Directory_Entry : in Directory_Entry_Type)
     return File_Kind;
```

118/2 Returns the kind of external file represented by Directory_Entry. The exception Status_Error is propagated if Directory_Entry is invalid.

119/2
```
function Size (Directory_Entry : in Directory_Entry_Type)
     return File_Size;
```

120/2 Returns the size of the external file represented by Directory_Entry. The size of an external file is the number of stream elements contained in the file. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Constraint_Error is propagated if the file size is not a value of type File_Size.

121/2
```
function Modification_Time (Directory_Entry : in Directory_Entry_Type)
     return Ada.Calendar.Time;
```

122/2 Returns the time that the external file represented by Directory_Entry was most recently modified. If the external file represented by Directory_Entry is not an ordinary file, the result is implementation defined. The exception Status_Error is propagated if Directory_Entry is invalid. The exception Use_Error is propagated if the external environment does not support reading the modification time of the file represented by Directory_Entry.

Implementation Requirements

For Copy_File, if Source_Name identifies an existing external ordinary file created by a predefined Ada input-output package, and Target_Name and Form can be used in the Create operation of that input-output package with mode Out_File without raising an exception, then Copy_File shall not propagate Use_Error. 123/2

Implementation Advice

If other information about a file (such as the owner or creation date) is available in a directory entry, the implementation should provide functions in a child package Directories.Information to retrieve it. 124/2

Start_Search and Search should raise Name_Error if Pattern is malformed, but not if it can represent a file in the directory but does not actually do so. 125/5

Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file. 126/2

NOTE 1 The operations Containing_Directory, Full_Name, Simple_Name, Base_Name, Extension, and Compose operate on file names, not external files. The files identified by these operations do not necessarily exist. Name_Error is raised only if the file name is malformed and cannot possibly identify a file. Of these operations, only the result of Full_Name depends on the current default directory; the result of the others depends only on their parameters. 127/5

NOTE 2 Using access types, values of Search_Type and Directory_Entry_Type can be saved and queried later. However, another task or application can modify or delete the file represented by a Directory_Entry_Type value or the directory represented by a Search_Type value; such a value can only give the information valid at the time it is created. Therefore, long-term storage of these values is not recommended. 128/2

NOTE 3 If the target system does not support directories inside of directories, then Kind will never return Directory and Containing_Directory will always raise Use_Error. 129/2

NOTE 4 If the target system does not support creation or deletion of directories, then Create_Directory, Create_Path, Delete_Directory, and Delete_Tree will always propagate Use_Error. 130/2

NOTE 5 To move a file or directory to a different location, use Rename. Most target systems will allow renaming of files from one directory to another. If the target file or directory can already exist, it should be deleted first. 131/5

A.16.1 The Package Directories.Hierarchical_File_Names

The library package Directories.Hierarchical_File_Names is an optional package providing operations for file name construction and decomposition for targets with hierarchical file naming. 1/3

Static Semantics

If provided, the library package Directories.Hierarchical_File_Names has the following declaration: 2/3

```
package Ada.Directories.Hierarchical_File_Names                                    3/5
   with Nonblocking, Global => in out synchronized is
   function Is_Simple_Name (Name : in String) return Boolean;                      4/3
   function Is_Root_Directory_Name (Name : in String) return Boolean;              5/3
   function Is_Parent_Directory_Name (Name : in String) return Boolean;            6/3
   function Is_Current_Directory_Name (Name : in String) return Boolean;           7/3
   function Is_Full_Name (Name : in String) return Boolean;                        8/3
   function Is_Relative_Name (Name : in String) return Boolean;                    9/3
   function Simple_Name (Name : in String) return String                           10/3
      renames Ada.Directories.Simple_Name;
   function Containing_Directory (Name : in String) return String                  11/3
      renames Ada.Directories.Containing_Directory;
   function Initial_Directory (Name : in String) return String;                    12/3
   function Relative_Name (Name : in String) return String;                        13/3
```

14/3
```
   function Compose (Directory      : in String := "";
                     Relative_Name  : in String;
                     Extension      : in String := "") return String;
```

15/3
```
end Ada.Directories.Hierarchical_File_Names;
```

16/3 In addition to the operations provided in package Directories.Hierarchical_File_Names, the operations in package Directories can be used with hierarchical file names. In particular, functions Full_Name, Base_Name, and Extension provide additional capabilities for hierarchical file names.

17/3
```
function Is_Simple_Name (Name : in String) return Boolean;
```

18/3 Returns True if Name is a simple name, and returns False otherwise.

19/3
```
function Is_Root_Directory_Name (Name : in String) return Boolean;
```

20/3 Returns True if Name is syntactically a root (a directory that cannot be decomposed further), and returns False otherwise.

21/3
```
function Is_Parent_Directory_Name (Name : in String) return Boolean;
```

22/3 Returns True if Name can be used to indicate symbolically the parent directory of any directory, and returns False otherwise.

23/3
```
function Is_Current_Directory_Name (Name : in String) return Boolean;
```

24/3 Returns True if Name can be used to indicate symbolically the directory itself for any directory, and returns False otherwise.

25/3
```
function Is_Full_Name (Name : in String) return Boolean;
```

26/3 Returns True if the leftmost directory part of Name is a root, and returns False otherwise.

27/3
```
function Is_Relative_Name (Name : in String) return Boolean;
```

28/3 Returns True if Name allows the identification of an external file (including directories and special files) but is not a full name, and returns False otherwise.

29/3
```
function Initial_Directory (Name : in String) return String;
```

30/3 Returns the leftmost directory part in Name. That is, it returns a root directory name (for a full name), or one of a parent directory name, a current directory name, or a simple name (for a relative name). The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files).

31/3
```
function Relative_Name (Name : in String) return String;
```

32/3 Returns the entire file name except the Initial_Directory portion. The exception Name_Error is propagated if the string given as Name does not allow the identification of an external file (including directories and special files), or if Name has a single part (this includes if any of Is_Simple_Name, Is_Root_Directory_Name, Is_Parent_Directory_Name, or Is_Current_Directory_Name are True).

33/3
```
function Compose (Directory      : in String := "";
                  Relative_Name  : in String;
                  Extension      : in String := "") return String;
```

34/3 Returns the name of the external file with the specified Directory, Relative_Name, and Extension. The exception Name_Error is propagated if the string given as Directory is not the null string and does not allow the identification of a directory, or if Is_Relative_Name (Relative_Name) is False, or if the string given as Extension is not the null string and is not a possible extension, or if Extension is not the null string and Simple_Name (Relative_Name) is not a base name.

The result of Compose is a full name if Is_Full_Name (Directory) is True; result is a relative name otherwise. 35/3

Implementation Advice

Directories.Hierarchical_File_Names should be provided for systems with hierarchical file naming, and should not be provided on other systems. 36/3

NOTE 1 These operations operate on file names, not external files. The files identified by these operations do not necessarily exist. Name_Error is raised only as specified or if the file name is malformed and cannot possibly identify a file. The result of these operations depends only on their parameters. 37/5

NOTE 2 Containing_Directory raises Use_Error if Name does not have a containing directory, including when any of Is_Simple_Name, Is_Root_Directory_Name, Is_Parent_Directory_Name, or Is_Current_Directory_Name are True. 38/3

A.16.2 The Packages Wide_Directories and Wide_Wide_Directories

The packages Wide_Directories and Wide_Wide_Directories provide operations for manipulating files and directories, and their names. 1/5

Static Semantics

The specification of package Wide_Directories is the same as for Directories (including its optional child packages Information and Hierarchical_File_Names), except that each occurrence of String is replaced by Wide_String. 2/5

The specification of package Wide_Wide_Directories is the same as for Directories (including its optional child packages Information and Hierarchical_File_Names), except that each occurrence of String is replaced by Wide_Wide_String. 3/5

A.17 The Package Environment_Variables

The package Environment_Variables allows a program to read or modify environment variables. Environment variables are name-value pairs, where both the name and value are strings. The definition of what constitutes an *environment variable*, and the meaning of the name and value, are implementation defined. 1/2

Static Semantics

The library package Environment_Variables has the following declaration: 2/2

```
package Ada.Environment_Variables                                                    3/5
   with Preelaborate, Nonblocking, Global => in out synchronized is

   function Value (Name : in String) return String;                                  4/2

   function Value (Name : in String; Default : in String) return String;             4.1/3

   function Exists (Name : in String) return Boolean;                                5/2

   procedure Set (Name : in String; Value : in String);                              6/2

   procedure Clear (Name : in String);                                               7/2
   procedure Clear;

   procedure Iterate                                                                 8/5
      (Process : not null access procedure (Name, Value : in String))
      with Allows_Exit;

end Ada.Environment_Variables;                                                       9/2
```

10/2 `function Value (Name : in String) return String;`

11/2 If the external execution environment supports environment variables, then Value returns the value of the environment variable with the given name. If no environment variable with the given name exists, then Constraint_Error is propagated. If the execution environment does not support environment variables, then Program_Error is propagated.

11.1/3 `function Value (Name : in String; Default : in String) return String;`

11.2/3 If the external execution environment supports environment variables and an environment variable with the given name currently exists, then Value returns its value; otherwise, it returns Default.

12/2 `function Exists (Name : in String) return Boolean;`

13/3 If the external execution environment supports environment variables and an environment variable with the given name currently exists, then Exists returns True; otherwise, it returns False.

14/2 `procedure Set (Name : in String; Value : in String);`

15/3 If the external execution environment supports environment variables, then Set first clears any existing environment variable with the given name, and then defines a single new environment variable with the given name and value. Otherwise, Program_Error is propagated.

16/2 If implementation-defined circumstances prohibit the definition of an environment variable with the given name and value, then Constraint_Error is propagated.

17/2 It is implementation defined whether there exist values for which the call Set(Name, Value) has the same effect as Clear (Name).

18/2 `procedure Clear (Name : in String);`

19/3 If the external execution environment supports environment variables, then Clear deletes all existing environment variables with the given name. Otherwise, Program_Error is propagated.

20/2 `procedure Clear;`

21/3 If the external execution environment supports environment variables, then Clear deletes all existing environment variables. Otherwise, Program_Error is propagated.

22/5
```
procedure Iterate
   (Process : not null access procedure (Name, Value : in String))
      with Allows_Exit;
```

23/3 If the external execution environment supports environment variables, then Iterate calls the subprogram designated by Process for each existing environment variable, passing the name and value of that environment variable. Otherwise, Program_Error is propagated.

24/2 If several environment variables exist that have the same name, Process is called once for each such variable.

Bounded (Run-Time) Errors

25/2 It is a bounded error to call Value if more than one environment variable exists with the given name; the possible outcomes are that:

26/2 - one of the values is returned, and that same value is returned in subsequent calls in the absence of changes to the environment; or

- Program_Error is propagated. 27/2

Erroneous Execution

Making calls to the procedures Set or Clear concurrently with calls to any subprogram of package Environment_Variables, or to any instantiation of Iterate, results in erroneous execution. 28/2

Making calls to the procedures Set or Clear in the actual subprogram corresponding to the Process parameter of Iterate results in erroneous execution. 29/2

Documentation Requirements

An implementation shall document how the operations of this package behave if environment variables are changed by external mechanisms (for instance, calling operating system services). 30/2

Implementation Permissions

An implementation running on a system that does not support environment variables is permitted to define the operations of package Environment_Variables with the semantics corresponding to the case where the external execution environment does support environment variables. In this case, it shall provide a mechanism to initialize a nonempty set of environment variables prior to the execution of a partition. 31/2

Implementation Advice

If the execution environment supports subprocesses, the currently defined environment variables should be used to initialize the environment variables of a subprocess. 32/2

Changes to the environment variables made outside the control of this package should be reflected immediately in the effect of the operations of this package. Changes to the environment variables made using this package should be reflected immediately in the external execution environment. This package should not perform any buffering of the environment variables. 33/2

A.17.1 The Packages Wide_Environment_Variables and Wide_Wide_Environment_Variables

The packages Wide_Environment_Variables and Wide_Wide_Environment_Variables allow a program to read or modify environment variables. 1/5

Static Semantics

The specification of package Wide_Environment_Variables is the same as for Environment_Variables, except that each occurrence of String is replaced by Wide_String. 2/5

The specification of package Wide_Wide_Environment_Variables is the same as for Environment_Variables, except that each occurrence of String is replaced by Wide_Wide_String. 3/5

A.18 Containers

This clause presents the specifications of the package Containers and several child packages, which provide facilities for storing collections of elements. 1/2

A variety of sequence and associative containers are provided. Each container package defines a *cursor* type as well as a container type. A cursor is a reference to an element within a container. Many operations on cursors are common to all of the containers. A cursor referencing an element in a container is considered to be overlapping only with the element itself. 2/5

2.1/5 Some operations of the language-defined child units of Ada.Containers have access-to-subprogram parameters. To ensure such operations are well-defined, they guard against certain actions by the designated subprogram. An action on a container that can add or remove an element is considered to *tamper with cursors*, and these are prohibited during all such operations. An action on a container that can replace an element with one of a different size is considered to *tamper with elements*, and these are prohibited during certain of such operations. The details of the specific actions that are considered to tamper with cursors or elements are defined for each child unit of Ada.Containers.

2.2/5 Several of the language-defined child units of Ada.Containers include a nested package named Stable, which provides a view of a container that prohibits any operations that would tamper with elements. By using a Stable view for manipulating a container, the number of tampering checks performed while performing the operations can be reduced. The details of the Stable subpackage are defined separately for each child unit of Ada.Containers that includes such a nested package.

3/2 Within this clause we provide Implementation Advice for the desired average or worst case time complexity of certain operations on a container. This advice is expressed using the Landau symbol $O(X)$. Presuming f is some function of a length parameter N and t(N) is the time the operation takes (on average or worst case, as specified) for the length N, a complexity of $O(f(N))$ means that there exists a finite A such that for any N, $t(N)/f(N) < A$.

4/2 If the advice suggests that the complexity should be less than $O(f(N))$, then for any arbitrarily small positive real D, there should exist a positive integer M such that for all $N > M$, $t(N)/f(N) < D$.

5/5 When a formal function is used to provide an ordering for a container, it is generally required to define a strict weak ordering. A function "<" defines a *strict weak ordering* if it is irreflexive, asymmetric, transitive, and in addition, if $x < y$ for any values x and y, then for all other values z, $(x < z)$ or $(z < y)$. Elements are in a *smallest first* order using such an operator if, for every element y with a predecessor x in the order, $(y < x)$ is false.

Static Semantics

6/5 Certain subprograms declared within instances of some of the generic packages presented in this clause are said to *perform indefinite insertion*. These subprograms are those corresponding (in the sense of the copying described in 12.3) to subprograms that have formal parameters of a generic formal indefinite type and that are identified as performing indefinite insertion in the subclause defining the generic package.

7/5 If a subprogram performs indefinite insertion, then certain run-time checks are performed as part of a call to the subprogram; if any of these checks fail, then the resulting exception is propagated to the caller and the container is not modified by the call. These checks are performed for each parameter corresponding (in the sense of the copying described in 12.3) to a parameter in the corresponding generic whose type is a generic formal indefinite type. The checks performed for a given parameter are those checks explicitly specified in 4.8 that would be performed as part of the evaluation of an initialized allocator whose access type is declared immediately within the instance, where:

8/4 - the value of the qualified_expression is that of the parameter; and

9/4 - the designated subtype of the access type is the subtype of the parameter; and

10/4 - finalization of the collection of the access type has started if and only if the finalization of the instance has started.

Implementation Requirements

11/5 For an indefinite container (one whose type is defined in an instance of a child package of Containers whose defining_identifier contains "Indefinite"), each element of the container shall be created when it is

inserted into the container and finalized when it is deleted from the container (or when the container object is finalized if the element has not been deleted). For a bounded container (one whose type is defined in an instance of a child package of Containers whose defining_identifier starts with "Bounded") that is not an indefinite container, all of the elements of the capacity of the container shall be created and default initialized when the container object is created; the elements shall be finalized when the container object is finalized. For other kinds of containers, when elements are created and finalized is unspecified.

For an instance *I* of a container package with a container type, the specific type *T* of the object returned from a function that returns an object of an iterator interface, as well as the primitive operations of *T*, shall be nonblocking. The Global aspect specified for *T* and the primitive operations of *T* shall be (**in all, out synchronized**) or a specification that allows access to fewer global objects. 12/5

A.18.1 The Package Containers

The package Containers is the root of the containers subsystem. 1/2

Static Semantics

The library package Containers has the following declaration: 2/2

```
package Ada.Containers                                                    3/5
   with Pure is
   type Hash_Type is mod implementation-defined;                          4/2
   type Count_Type is range 0 .. implementation-defined;                  5/2
   Capacity_Error : exception;                                            5.1/3
end Ada.Containers;                                                       6/2
```

Hash_Type represents the range of the result of a hash function. Count_Type represents the (potential or actual) number of elements of a container. 7/2

Capacity_Error is raised when the capacity of a container is exceeded. 7.1/3

Implementation Advice

Hash_Type'Modulus should be at least 2**32. Count_Type'Last should be at least 2**31–1. 8/2

A.18.2 The Generic Package Containers.Vectors

The language-defined generic package Containers.Vectors provides private types Vector and Cursor, and a set of operations for each type. A vector container allows insertion and deletion at any position, but it is specifically optimized for insertion and deletion at the high end (the end with the higher index) of the container. A vector container also provides random access to its elements. 1/2

A vector container behaves conceptually as an array that expands as necessary as items are inserted. The *length* of a vector is the number of elements that the vector contains. The *capacity* of a vector is the maximum number of elements that can be inserted into the vector prior to it being automatically expanded. 2/2

Elements in a vector container can be referred to by an index value of a generic formal type. The first element of a vector always has its index value equal to the lower bound of the formal type. 3/2

A vector container may contain *empty elements*. Empty elements do not have a specified value. 4/2

Static Semantics

5/2 The generic library package Containers.Vectors has the following declaration:

6/5
```
with Ada.Iterator_Interfaces;
generic
   type Index_Type is range <>;
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Vectors
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

7/2
```
   subtype Extended_Index is
      Index_Type'Base range
         Index_Type'First-1 ..
         Index_Type'Min (Index_Type'Base'Last - 1, Index_Type'Last) + 1;
   No_Index : constant Extended_Index := Extended_Index'First;
```

8/5
```
   type Vector is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Vector,
           Aggregate         => (Empty           => Empty,
                                 Add_Unnamed     => Append,
                                 New_Indexed     => New_Vector,
                                 Assign_Indexed => Replace_Element),
           Stable_Properties => (Length, Capacity,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (Vector) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Vector)) and then
              (not Tampering_With_Elements_Prohibited (Vector)),
           Preelaborable_Initialization;
```

9/5
```
   type Cursor is private
      with Preelaborable_Initialization;
```

10/2
```
   Empty_Vector : constant Vector;
```

11/2
```
   No_Element : constant Cursor;
```

11.1/5
```
   function Has_Element (Position : Cursor) return Boolean
      with Nonblocking, Global => in all, Use_Formal => null;
```

11.2/5
```
   function Has_Element (Container : Vector; Position : Cursor)
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

11.3/5
```
   package Vector_Iterator_Interfaces is new
       Ada.Iterator_Interfaces (Cursor, Has_Element);
```

12/2
```
   function "=" (Left, Right : Vector) return Boolean;
```

12.1/5
```
   function Tampering_With_Cursors_Prohibited
      (Container : Vector) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

12.2/5
```
   function Tampering_With_Elements_Prohibited
      (Container : Vector) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

12.3/5
```
   function Maximum_Length return Count_Type
      with Nonblocking, Global => null, Use_Formal => null;
```

```
function Empty (Capacity : Count_Type := implementation-defined)                    12.4/5
   return Vector
   with Pre  => Capacity <= Maximum_Length
                   or else raise Constraint_Error,
        Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Elements_Prohibited (Empty'Result) and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;

function To_Vector (Length : Count_Type) return Vector                               13/5
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
           To_Vector'Result.Capacity >= Length;

function To_Vector                                                                   14/5
  (New_Item : Element_Type;
   Length   : Count_Type) return Vector
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
           To_Vector'Result.Capacity >= Length;

function New_Vector (First, Last : Index_Type) return Vector is                    14.1/5
     (To_Vector (Count_Type (Last - First + 1)))
   with Pre => First = Index_Type'First;

function "&" (Left, Right : Vector) return Vector                                    15/5
   with Pre  => Length (Left) <= Maximum_Length - Length (Right)
                   or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) =
                  Length (Left) + Length (Right) and then
                not Tampering_With_Elements_Prohibited
                  (Vectors."&"'Result) and then
                not Tampering_With_Cursors_Prohibited
                  (Vectors."&"'Result) and then
                Vectors."&"'Result.Capacity >=
                  Length (Left) + Length (Right);

function "&" (Left  : Vector;                                                        16/5
              Right : Element_Type) return Vector
   with Pre  => Length (Left) <= Maximum_Length - 1
                   or else raise Constraint_Error,
        Post => Vectors."&"'Result.Length = Length (Left) + 1 and then
                not Tampering_With_Elements_Prohibited
                  (Vectors."&"'Result) and then
                not Tampering_With_Cursors_Prohibited
                  (Vectors."&"'Result) and then
                Vectors."&"'Result.Capacity >= Length (Left) + 1;

function "&" (Left  : Element_Type;                                                  17/5
              Right : Vector) return Vector
   with Pre  => Length (Right) <= Maximum_Length - 1
                   or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) = Length (Right) + 1 and then
                not Tampering_With_Elements_Prohibited
                  (Vectors."&"'Result) and then
                not Tampering_With_Cursors_Prohibited
                  (Vectors."&"'Result) and then
                Vectors."&"'Result.Capacity >= Length (Right) + 1;
```

```
18/5        function "&" (Left, Right  : Element_Type) return Vector
               with Pre  => Maximum_Length >= 2 or else raise Constraint_Error,
                    Post => Length ("&"'Result) = 2 and then
                            not Tampering_With_Elements_Prohibited
                              (Vectors."&"'Result) and then
                            not Tampering_With_Cursors_Prohibited
                              (Vectors."&"'Result) and then
                            Vectors."&"'Result.Capacity >= 2;

19/5        function Capacity (Container : Vector) return Count_Type
               with Nonblocking, Global => null, Use_Formal => null;

20/5        procedure Reserve_Capacity (Container : in out Vector;
                                        Capacity  : in     Count_Type)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                               or else raise Program_Error,
                    Post => Container.Capacity >= Capacity;

21/5        function Length (Container : Vector) return Count_Type
               with Nonblocking, Global => null, Use_Formal => null;

22/5        procedure Set_Length (Container : in out Vector;
                                  Length    : in     Count_Type)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                               or else raise Program_Error) and then
                            (Length <= Maximum_Length
                               or else raise Constraint_Error),
                    Post => Container.Length = Length and then
                            Capacity (Container) >= Length;

23/5        function Is_Empty (Container : Vector) return Boolean
               with Nonblocking, Global => null, Use_Formal => null,
                    Post => Is_Empty'Result = (Length (Container) = 0);

24/5        procedure Clear (Container : in out Vector)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                              or else raise Program_Error,
                    Post => Length (Container) = 0;

25/5        function To_Cursor (Container : Vector;
                                Index     : Extended_Index) return Cursor
               with Post => (if Index in
                               First_Index (Container) .. Last_Index (Container)
                             then Has_Element (Container, To_Cursor'Result)
                             else To_Cursor'Result = No_Element),
                    Nonblocking, Global => null, Use_Formal => null;

26/5        function To_Index (Position  : Cursor) return Extended_Index
               with Nonblocking, Global => in all;

26.1/5      function To_Index (Container : Vector;
                               Position  : Cursor) return Extended_Index
               with Pre  => Position = No_Element or else
                            Has_Element (Container, Position) or else
                               raise Program_Error,
                    Post => (if Position = No_Element then To_Index'Result = No_Index
                             else To_Index'Result in First_Index (Container) ..
                                  Last_Index (Container)),
                    Nonblocking, Global => null, Use_Formal => null;

27/5        function Element (Container : Vector;
                              Index     : Index_Type)
               return Element_Type
               with Pre  => Index in
                               First_Index (Container) .. Last_Index (Container)
                            or else raise Constraint_Error,
                    Nonblocking, Global => null, Use_Formal => Element_Type;

28/5        function Element (Position : Cursor) return Element_Type
               with Pre  => Position /= No_Element or else raise Constraint_Error,
                    Nonblocking, Global => in all, Use_Formal => Element_Type;
```

```
function Element (Container : Vector;                                    28.1/5
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element or else
                   raise Constraint_Error) and then
               (Has_Element (Container, Position)
                  or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;

procedure Replace_Element (Container : in out Vector;                    29/5
                           Index     : in     Index_Type;
                           New_Item  : in     Element_Type)
   with Pre => (not Tampering_With_Elements_Prohibited (Container)
                  or else raise Program_Error) and then
               (Index in
                  First_Index (Container) .. Last_Index (Container)
                  or else raise Constraint_Error);

procedure Replace_Element (Container : in out Vector;                    30/5
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                  or else raise Program_Error) and then
                (Position /= No_Element
                  or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                  or else raise Program_Error);

procedure Query_Element                                                  31/5
  (Container : in Vector;
   Index     : in Index_Type;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => Index in
                   First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error;

procedure Query_Element                                                  32/5
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;

procedure Query_Element                                                  32.1/5
  (Container : in Vector;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);

procedure Update_Element                                                 33/5
  (Container : in out Vector;
   Index     : in     Index_Type;
   Process   : not null access procedure
                  (Element : in out Element_Type))
   with Pre  => Index in
                   First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error;

procedure Update_Element                                                 34/5
  (Container : in out Vector;
   Position  : in     Cursor;
   Process   : not null access procedure
                   (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

34.1/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
       limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

34.2/6
```
type Reference_Type
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

34.3/5
```
function Constant_Reference (Container : aliased in Vector;
                             Index     : in Index_Type)
   return Constant_Reference_Type
   with Pre    => Index in
                    First_Index (Container) .. Last_Index (Container)
                    or else raise Constraint_Error,
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

34.4/5
```
function Reference (Container : aliased in out Vector;
                    Index     : in Index_Type)
   return Reference_Type
   with Pre    => Index in
                    First_Index (Container) .. Last_Index (Container)
                    or else raise Constraint_Error,
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

34.5/5
```
function Constant_Reference (Container : aliased in Vector;
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

34.6/5
```
function Reference (Container : aliased in out Vector;
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

34.7/5
```
procedure Assign (Target : in out Vector; Source : in Vector)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target) and then
                Capacity (Target) >= Length (Target);
```

34.8/5
```
function Copy (Source : Vector; Capacity : Count_Type := 0)
   return Vector
   with Pre => Capacity = 0 or else Capacity >= Length (Source)
                    or else raise Capacity_Error,
        Post => Length (Copy'Result) = Length (Source) and then
                not Tampering_With_Elements_Prohibited (Copy'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Copy'Result)
                   and then
                Copy'Result.Capacity >= (if Capacity = 0 then
                   Length (Source) else Capacity);
```

```
procedure Move (Target : in out Vector;                                    35/5
                Source : in out Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Length (Target) = Length (Source)'Old and then
                    Length (Source) = 0 and then
                    Capacity (Target) >= Length (Source)'Old);

procedure Insert_Vector (Container : in out Vector;                        36/5
                         Before    : in     Extended_Index;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);

procedure Insert_Vector (Container : in out Vector;                        37/5
                         Before    : in     Cursor;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);

procedure Insert_Vector (Container : in out Vector;                        38/5
                         Before    : in     Cursor;
                         New_Item  : in     Vector;
                         Position  :    out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);

procedure Insert (Container : in out Vector;                               39/5
                  Before    : in     Extended_Index;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);
```

40/5
```
      procedure Insert (Container : in out Vector;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                          or else raise Program_Error) and then
                       (Before = No_Element or else
                        Has_Element (Container, Before)
                          or else raise Program_Error) and then
                       (Length (Container) <= Maximum_Length - Count
                          or else raise Constraint_Error),
              Post => Length (Container)'Old + Count =
                          Length (Container) and then
                      Capacity (Container) >= Length (Container);
```

41/5
```
      procedure Insert (Container : in out Vector;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                          or else raise Program_Error) and then
                       (Before = No_Element or else
                        Has_Element (Container, Before)
                          or else raise Program_Error) and then
                       (Length (Container) <= Maximum_Length - Count
                          or else raise Constraint_Error),
              Post => Length (Container)'Old + Count =
                          Length (Container) and then
                      Has_Element (Container, Position) and then
                      Capacity (Container) >= Length (Container);
```

42/5
```
      procedure Insert (Container : in out Vector;
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                          or else raise Program_Error) and then
                       (Before in
                          First_Index (Container) .. Last_Index (Container) + 1
                          or else raise Constraint_Error) and then
                       (Length (Container) <= Maximum_Length - Count
                          or else raise Constraint_Error),
              Post => Length (Container)'Old + Count =
                          Length (Container) and then
                      Capacity (Container) >= Length (Container);
```

43/5
```
      procedure Insert (Container : in out Vector;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                          or else raise Program_Error) and then
                       (Before = No_Element or else
                        Has_Element (Container, Before)
                          or else raise Program_Error) and then
                       (Length (Container) <= Maximum_Length - Count
                          or else raise Constraint_Error),
              Post => Length (Container)'Old + Count = Length (Container)
                      and then Has_Element (Container, Position) and then
                      Capacity (Container) >= Length (Container);
```

```
procedure Prepend_Vector (Container : in out Vector;                                    44/5
                          New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                    Length (Container) and then
                Capacity (Container) >= Length (Container);

procedure Prepend (Container : in out Vector;                                           45/5
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count =
                    Length (Container) and then
                Capacity (Container) >= Length (Container);

procedure Append_Vector (Container : in out Vector;                                     46/5
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                    Length (Container) and then
                Capacity (Container) >= Length (Container);

procedure Append (Container : in out Vector;                                            47/5
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                     or else raise Constraint_Error),
        Post =>
           Length (Container)'Old + Count = Length (Container) and then
           Capacity (Container) >= Length (Container);

procedure Append (Container : in out Vector;                                            47.1/5
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - 1
                     or else raise Constraint_Error),
        Post => Length (Container)'Old + 1 = Length (Container) and then
                Capacity (Container) >= Length (Container);

procedure Insert_Space (Container : in out Vector;                                      48/5
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);
```

```
49/5        procedure Insert_Space (Container : in out Vector;
                                    Before    : in     Cursor;
                                    Position  :    out Cursor;
                                    Count     : in     Count_Type := 1)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                 or else raise Program_Error) and then
                             (Before = No_Element or else
                                 Has_Element (Container, Before)
                                 or else raise Program_Error) and then
                             (Length (Container) <= Maximum_Length - Count
                                 or else raise Constraint_Error),
                    Post => Length (Container)'Old + Count =
                                 Length (Container) and then
                            Has_Element (Container, Position) and then
                            Capacity (Container) >= Length (Container);

50/5        procedure Delete (Container : in out Vector;
                              Index     : in     Extended_Index;
                              Count     : in     Count_Type := 1)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                 or else raise Program_Error) and then
                             (Index in
                                 First_Index (Container) .. Last_Index (Container) + 1
                                 or else raise Constraint_Error),
                    Post => Length (Container)'Old - Count <=
                                 Length (Container);

51/5        procedure Delete (Container : in out Vector;
                              Position  : in out Cursor;
                              Count     : in     Count_Type := 1)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                 or else raise Program_Error) and then
                             (Position /= No_Element
                                 or else raise Constraint_Error) and then
                             (Has_Element (Container, Position)
                                 or else raise Program_Error),
                    Post => Length (Container)'Old - Count <=
                                 Length (Container) and then
                            Position = No_Element;

52/5        procedure Delete_First (Container : in out Vector;
                                    Count     : in     Count_Type := 1)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error,
                    Post => Length (Container)'Old - Count <= Length (Container);

53/5        procedure Delete_Last (Container : in out Vector;
                                   Count     : in     Count_Type := 1)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error,
                    Post => Length (Container)'Old - Count <= Length (Container);

53.1/6      function Slice (Container : Vector;
                            Low       : Index_Type;
                            High      : Extended_Index) return Vector
               with Pre  => High in Low - 1 .. Container.Last_Index
                                or else raise Constraint_Error,
                    Post => Slice'Result.Length = High - Low + 1 and then
                            not Tampering_With_Elements_Prohibited (Slice'Result)
                                and then
                            not Tampering_With_Cursors_Prohibited (Slice'Result)
                                and then
                            Slice'Result.Capacity >= Slice'Result.Length;
```

```
procedure Slice (Container : in out Vector;                                    53.2/6
                 Low       : in     Index_Type;
                 High      : in     Extended_Index)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (High in Low - 1 .. Container.Last_Index
                    or else raise Constraint_Error),
        Post => Container.Length = High - Low + 1 and then
                Container.Capacity >= Container.Length;

function Replace_Slice (Container : Vector;                                    53.3/6
                        Low       : Index_Type;
                        High      : Extended_Index;
                        By        : Vector) return Vector
   with Pre  => (High in Low - 1 .. Container.Last_Index
                    or else raise Constraint_Error) and then
                 (Container.Length - Count_Type (High - Low + 1) <=
                    Maximum_Length - By.Length
                    or else raise Constraint_Error),
        Post => Replace_Slice'Result.Length = Container.Length -
                    Count_Type (High - Low + 1) + By.Length and then
                not Tampering_With_Elements_Prohibited
                    (Replace_Slice'Result) and then
                not Tampering_With_Cursors_Prohibited
                    (Replace_Slice'Result) and then
                Replace_Slice'Result.Capacity >= Replace_Slice.Length;

procedure Replace_Slice (Container : in out Vector;                            53.4/6
                         Low       : in     Index_Type;
                         High      : in     Extended_Index;
                         By        : in     Vector)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (High in Low - 1 .. Container.Last_Index
                    or else raise Constraint_Error) and then
                 (Container.Length - Count_Type (High - Low + 1) <=
                    Maximum_Length - By.Length
                    or else raise Constraint_Error),
        Post => Container.Length = Container.Length'Old -
                    Count_Type (High - Low + 1) + By.Length and then
                Container.Capacity >= Container.Length;

procedure Reverse_Elements (Container : in out Vector)                         54/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;

procedure Swap (Container : in out Vector;                                     55/5
                I, J      : in     Index_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (I in First_Index (Container) .. Last_Index (Container)
                    or else raise Constraint_Error) and then
                (J in First_Index (Container) .. Last_Index (Container)
                    or else raise Constraint_Error);

procedure Swap (Container : in out Vector;                                     56/5
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                    or else raise Program_Error) and then
                (Has_Element (Container, J)
                    or else raise Program_Error);

function First_Index (Container : Vector) return Index_Type                    57/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => First_Index'Result = Index_Type'First;
```

58/5
```
function First (Container : Vector) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

59/5
```
function First_Element (Container : Vector)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                  or else raise Constraint_Error);
```

60/5
```
function Last_Index (Container : Vector) return Extended_Index
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if Length (Container) = 0
                 then Last_Index'Result = No_Index
                 else Count_Type(Last_Index'Result - Index_Type'First) =
                     Length (Container) - 1);
```

61/5
```
function Last (Container : Vector) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

62/5
```
function Last_Element (Container : Vector)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                  or else raise Constraint_Error);
```

63/5
```
function Next (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

63.1/5
```
function Next (Container : Vector; Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Has_Element (Container, Next'Result) then
                    To_Index (Container, Next'Result) =
                    To_Index (Container, Position) + 1
                 elsif Next'Result = No_Element then
                    Position = Last (Container)
                 else False);
```

64/5
```
procedure Next (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

64.1/5
```
procedure Next (Container : in     Vector;
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

65/5
```
function Previous (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                 then Previous'Result = No_Element);
```

```
   function Previous (Container : Vector;                                    65.1/5
                      Position  : Cursor) return Cursor
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position = No_Element
                    then Previous'Result = No_Element
                    elsif Has_Element (Container, Previous'Result) then
                       To_Index (Container, Previous'Result) =
                       To_Index (Container, Position) - 1
                    elsif Previous'Result = No_Element then
                       Position = First (Container)
                    else False);

   procedure Previous (Position : in out Cursor)                             66/5
      with Nonblocking, Global => in all, Use_Formal => null;

   procedure Previous (Container : in     Vector;                            66.1/5
                       Position  : in out Cursor)
      with Nonblocking, Global => null, Use_Formal => null,
           Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Position /= No_Element
                    then Has_Element (Container, Position));

   function Find_Index (Container : Vector;                                  67/2
                        Item      : Element_Type;
                        Index     : Index_Type := Index_Type'First)
      return Extended_Index;

   function Find (Container : Vector;                                        68/5
                  Item      : Element_Type;
                  Position  : Cursor := No_Element)
      return Cursor
      with Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));

   function Reverse_Find_Index (Container : Vector;                          69/2
                                Item      : Element_Type;
                                Index     : Index_Type := Index_Type'Last)
      return Extended_Index;

   function Reverse_Find (Container : Vector;                                70/5
                          Item      : Element_Type;
                          Position  : Cursor := No_Element)
      return Cursor
      with Pre  => Position = No_Element or else
                   Has_Element (Container, Position)
                      or else raise Program_Error,
           Post => (if Reverse_Find'Result /= No_Element
                    then Has_Element (Container, Reverse_Find'Result));

   function Contains (Container : Vector;                                    71/2
                      Item      : Element_Type) return Boolean;
```

This paragraph was deleted. 72/3

```
   procedure Iterate                                                         73/5
     (Container : in Vector;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;

   procedure Reverse_Iterate                                                 74/5
     (Container : in Vector;
      Process   : not null access procedure (Position : in Cursor))
      with Allows_Exit;
```

```
74.1/5          function Iterate (Container : in Vector)
                   return Vector_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
                   with Post   => Tampering_With_Cursors_Prohibited (Container);
74.2/5          function Iterate (Container : in Vector; Start : in Cursor)
                   return Vector_Iterator_Interfaces.Reversible_Iterator'Class
                   with Pre    => (Start /= No_Element
                                      or else raise Constraint_Error) and then
                                  (Has_Element (Container, Start)
                                      or else raise Program_Error),
                        Post   => Tampering_With_Cursors_Prohibited (Container);
75/5            generic
                   with function "<" (Left, Right : Element_Type)
                      return Boolean is <>;
                package Generic_Sorting
                with Nonblocking, Global => null is
76/2               function Is_Sorted (Container : Vector) return Boolean;
77/5               procedure Sort (Container : in out Vector)
                      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                      or else raise Program_Error;
78/5               procedure Merge (Target  : in out Vector;
                                    Source  : in out Vector)
                      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                                      or else raise Program_Error) and then
                                   (not Tampering_With_Cursors_Prohibited (Source)
                                      or else raise Program_Error) and then
                                   (Length (Target) <= Maximum_Length - Length (Source)
                                      or else raise Constraint_Error) and then
                                   ((Length (Source) = 0 or else
                                      not Target'Has_Same_Storage (Source))
                                      or else raise Program_Error),
                           Post => (declare
                                      Result_Length : constant Count_Type :=
                                         Length (Source)'Old + Length (Target)'Old;
                                    begin
                                      (Length (Source) = 0 and then
                                       Length (Target) = Result_Length and then
                                       Capacity (Target) >= Result_Length));
79/2            end Generic_Sorting;
79.1/5          package Stable is
79.2/5             type Vector (Base : not null access Vectors.Vector) is
                      tagged limited private
                      with Constant_Indexing => Constant_Reference,
                           Variable_Indexing => Reference,
                           Default_Iterator  => Iterate,
                           Iterator_Element  => Element_Type,
                           Stable_Properties => (Length, Capacity),
                           Global            => null,
                           Default_Initial_Condition => Length (Vector) = 0,
                           Preelaborable_Initialization;
79.3/5             type Cursor is private
                      with Preelaborable_Initialization;
79.4/5             Empty_Vector : constant Vector;
79.5/5             No_Element : constant Cursor;
79.6/5             function Has_Element (Position : Cursor) return Boolean
                      with Nonblocking, Global => in all, Use_Formal => null;
79.7/5             package Vector_Iterator_Interfaces is new
                      Ada.Iterator_Interfaces (Cursor, Has_Element);
```

```
      procedure Assign (Target : in out Vectors.Vector;                                    79.8/5
                        Source : in Vector)
         with Post => Length (Source) = Length (Target) and then
                      Capacity (Target) >= Length (Target);
      function Copy (Source : Vectors.Vector) return Vector                                79.9/5
         with Post => Length (Copy'Result) = Length (Source);
      type Constant_Reference_Type                                                         79.10/6
            (Element : not null access constant Element_Type) is
               limited private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);
      type Reference_Type                                                                  79.11/6
            (Element : not null access Element_Type) is limited private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);
      -- Additional subprograms as described in the text                                   79.12/5
      -- are declared here.
   private                                                                                 79.13/5
      ... -- not specified by the language                                                 79.14/5
   end Stable;                                                                             79.15/5
private                                                                                    80/2
   ... -- not specified by the language                                                    81/2
end Ada.Containers.Vectors;                                                                82/2
```

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions defined to use it return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions defined to use it are unspecified. 83/2

The type Vector is used to represent vectors. The type Vector needs finalization (see 7.6). 84/2

Empty_Vector represents the empty vector object. It has a length of 0. If an object of type Vector is not otherwise initialized, it is initialized to the same value as Empty_Vector. 85/2

No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element. 86/2

The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container. 87/5

Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error. 88/2

Vector'Write for a Vector object *V* writes Length(*V*) elements of the vector to the stream. It may also write additional information about the vector. 88.1/5

Vector'Read reads the representation of a vector from the stream, and assigns to *Item* a vector with the same length and elements as was written by Vector'Write. 88.2/3

No_Index represents a position that does not correspond to any element. The subtype Extended_Index includes the indices covered by Index_Type plus the value No_Index and, if it exists, the successor to the Index_Type'Last. 89/2

90/5 Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced. When tampering with cursors is *prohibited* for a particular vector object *V*, Program_Error is propagated by the finalization of *V*, as well as by a call that passes *V* to certain of the operations of this package, as indicated by the precondition of such an operation. Similarly, when tampering with elements is *prohibited* for *V*, Program_Error is propagated by a call that passes *V* to certain of the other operations of this package, as indicated by the precondition of such an operation.

Paragraphs 91 through 97 are removed as preconditions now describe these rules.

97.2/5
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

97.3/3 Returns True if Position designates an element, and returns False otherwise.

97.4/5
```
function Has_Element (Container : Vector; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

97.5/5 Returns True if Position designates an element in Container, and returns False otherwise.

98/2
```
function "=" (Left, Right : Vector) return Boolean;
```

99/3 If Left and Right denote the same vector object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any such comparison returns False, the function returns False; otherwise, it returns True. Any exception raised during evaluation of element equality is propagated.

99.1/5
```
function Tampering_With_Cursors_Prohibited
   (Container : Vector) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

99.2/5 Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise.

99.3/5
```
function Tampering_With_Elements_Prohibited
   (Container : Vector) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

99.4/5 Always returns False, regardless of whether tampering with elements is prohibited.

99.5/5
```
function Maximum_Length return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

99.6/5 Returns the maximum Length of a Vector, based on the index type.

99.7/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Vector
   with Pre  => Capacity <= Maximum_Length
                   or else raise Constraint_Error,
        Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Elements_Prohibited (Empty'Result) and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;
```

99.8/5 Returns an empty vector.

```
function To_Vector (Length : Count_Type) return Vector                                  100/5
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
           To_Vector'Result.Capacity >= Length;
```

Returns a vector with a length of Length, filled with empty elements. 101/2

```
function To_Vector                                                                      102/5
  (New_Item : Element_Type;
   Length   : Count_Type) return Vector
   with Pre  => Length <= Maximum_Length or else raise Constraint_Error,
        Post =>
           To_Vector'Result.Length = Length and then
           not Tampering_With_Elements_Prohibited (To_Vector'Result)
             and then
           not Tampering_With_Cursors_Prohibited (To_Vector'Result)
             and then
        To_Vector'Result.Capacity >= Length;
```

Returns a vector with a length of Length, filled with elements initialized to the value New_Item. 103/2

```
function "&" (Left, Right : Vector) return Vector                                       104/5
   with Pre  => Length (Left) <= Maximum_Length - Length (Right)
                 or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) =
                   Length (Left) + Length (Right) and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >=
                   Length (Left) + Length (Right);
```

Returns a vector comprising the elements of Left followed by the elements of Right. 105/2

```
function "&" (Left  : Vector;                                                           106/5
              Right : Element_Type) return Vector
   with Pre  => Length (Left) <= Maximum_Length - 1
                   or else raise Constraint_Error,
        Post => Vectors."&"'Result.Length = Length (Left) + 1 and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >= Length (Left) + 1;
```

Returns a vector comprising the elements of Left followed by the element Right. 107/2

```
function "&" (Left  : Element_Type;                                                     108/5
              Right : Vector) return Vector
   with Pre  => Length (Right) <= Maximum_Length - 1
                   or else raise Constraint_Error,
        Post => Length (Vectors."&"'Result) = Length (Right) + 1 and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >= Length (Right) + 1;
```

Returns a vector comprising the element Left followed by the elements of Right. 109/2

110/5
```
function "&" (Left, Right  : Element_Type) return Vector
   with Pre  => Maximum_Length >= 2 or else raise Constraint_Error,
        Post => Length ("&"'Result) = 2 and then
                not Tampering_With_Elements_Prohibited (Vectors."&"'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Vectors."&"'Result)
                   and then
                Vectors."&"'Result.Capacity >= 2;
```

111/2 Returns a vector comprising the element Left followed by the element Right.

112/5
```
function Capacity (Container : Vector) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

113/2 Returns the capacity of Container.

114/5
```
procedure Reserve_Capacity (Container : in out Vector;
                            Capacity  : in     Count_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                 or else raise Program_Error,
        Post => Container.Capacity >= Capacity;
```

115/3 If the capacity of Container is already greater than or equal to Capacity, then Reserve_Capacity has no effect. Otherwise, Reserve_Capacity allocates additional storage as necessary to ensure that the length of the resulting vector can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then, as necessary, moves elements into the new storage and deallocates any storage no longer needed. Any exception raised during allocation is propagated and Container is not modified.

116/5
```
function Length (Container : Vector) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

117/2 Returns the number of elements in Container.

118/5
```
procedure Set_Length (Container : in out Vector;
                      Length    : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length <= Maximum_Length or else raise Constraint_Error),
        Post => Container.Length = Length and then
                Capacity (Container) >= Length;
```

119/3 If Length is larger than the capacity of Container, Set_Length calls Reserve_Capacity (Container, Length), then sets the length of the Container to Length. If Length is greater than the original length of Container, empty elements are added to Container; otherwise, elements are removed from Container.

120/5
```
function Is_Empty (Container : Vector) return Boolean
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```

121/5 Returns True if Container is empty.

122/5
```
procedure Clear (Container : in out Vector)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

123/2 Removes all the elements from Container. The capacity of Container does not change.

```
function To_Cursor (Container : Vector;                                    124/5
                    Index     : Extended_Index) return Cursor
   with Post => (if Index in
                    First_Index (Container) .. Last_Index (Container)
                 then Has_Element (Container, To_Cursor'Result)
                 else To_Cursor'Result = No_Element),
        Nonblocking, Global => null, Use_Formal => null;
```

Returns a cursor designating the element at position Index in Container; returns No_Element if Index does not designate an element. For the purposes of determining whether the parameters overlap in a call to To_Cursor, the Container parameter is not considered to overlap with any object (including itself). 125/5

```
function To_Index (Position  : Cursor) return Extended_Index               126/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

If Position is No_Element, No_Index is returned. Otherwise, the index (within its containing vector) of the element designated by Position is returned. 127/2

```
function To_Index (Container : Vector;                                     127.1/5
                   Position  : Cursor) return Extended_Index
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)  or else
                   raise Program_Error,
        Post => (if Position = No_Element then To_Index'Result = No_Index
                 else To_Index'Result in First_Index (Container) ..
                         Last_Index (Container)),
        Nonblocking, Global => null, Use_Formal => null;
```

Returns the index (within Container) of the element designated by Position; returns No_Index if Position does not designate an element. For the purposes of determining whether the parameters overlap in a call to To_Index, the Container parameter is not considered to overlap with any object (including itself). 127.2/5

```
function Element (Container : Vector;                                      128/5
                  Index     : Index_Type)
   return Element_Type
   with Pre  => Index in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error,
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

Element returns the element at position Index. 129/5

```
function Element (Position  : Cursor) return Element_Type                  130/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

Element returns the element designated by Position. 131/5

```
function Element (Container : Vector;                                      131.1/5
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

Element returns the element designated by Position in Container. 131.2/5

132/5
```
procedure Replace_Element (Container : in out Vector;
                           Index     : in     Index_Type;
                           New_Item  : in     Element_Type)
   with Pre => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
               (Index in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error);
```

133/5 Replace_Element assigns the value New_Item to the element at position Index. Any exception raised during the assignment is propagated. The element at position Index is not an empty element after successful call to Replace_Element. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object (including itself), and the Index parameter is considered to overlap with the element at position Index.

134/5
```
procedure Replace_Element (Container : in out Vector;
                           Position  : in     Cursor;
                           New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
               (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error);
```

135/5 Replace_Element assigns New_Item to the element designated by Position. Any exception raised during the assignment is propagated. The element at Position is not an empty element after successful call to Replace_Element. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object (including itself).

136/5
```
procedure Query_Element
  (Container : in Vector;
   Index     : in Index_Type;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => Index in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error;
```

137/5 Query_Element calls Process.**all** with the element at position Index as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

138/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element or else raise Constraint_Error
        Global => in all;
```

139/5 Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the vector that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

139.1/5
```
procedure Query_Element
  (Container : in Vector;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error);
```

Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 139.2/5

```
procedure Update_Element                                                    140/5
   (Container : in out Vector;
    Index     : in     Index_Type;
    Process   : not null access procedure
                      (Element : in out Element_Type))
    with Pre  => Index in First_Index (Container) .. Last_Index (Container)
                      or else raise Constraint_Error;
```

Update_Element calls Process.**all** with the element at position Index as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 141/5

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained. 142/2

The element at position Index is not an empty element after successful completion of this operation. 143/2

```
procedure Update_Element                                                    144/5
   (Container : in out Vector;
    Position  : in     Cursor;
    Process   : not null access procedure
                      (Element : in out Element_Type))
    with Pre  => (Position /= No_Element
                      or else raise Constraint_Error) and then
                  (Has_Element (Container, Position)
                      or else raise Program_Error);
```

Update_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 145/5

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained. 146/2

The element designated by Position is not an empty element after successful completion of this operation. 147/2

```
type Constant_Reference_Type                                                147.1/6
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
type Reference_Type                                                         147.2/6
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

The types Constant_Reference_Type and Reference_Type need finalization. 147.3/3

This paragraph was deleted. 147.4/5

147.5/5
```
function Constant_Reference (Container : aliased in Vector;
                             Index     : in Index_Type)
   return Constant_Reference_Type
   with Pre    => Index in First_Index (Container) .. Last_Index (Container)
                    or else raise Constraint_Error,
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

147.6/3 This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a vector given an index value.

147.7/5 Constant_Reference returns an object whose discriminant is an access value that designates the element at position Index. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

147.8/5
```
function Reference (Container : aliased in out Vector;
                    Index     : in Index_Type)
   return Reference_Type
   with Pre    => Index in First_Index (Container) .. Last_Index (Container)
                    or else raise Constraint_Error,
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

147.9/3 This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a vector given an index value.

147.10/5 Reference returns an object whose discriminant is an access value that designates the element at position Index. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized.

147.11/3 The element at position Index is not an empty element after successful completion of this operation.

147.12/5
```
function Constant_Reference (Container : aliased in Vector;
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

147.13/3 This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a vector given a cursor.

147.14/5 Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

```
function Reference (Container : aliased in out Vector;                    147.15/5
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a vector given a cursor. 147.16/3

Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized. 147.17/5

The element designated by Position is not an empty element after successful completion of this operation. 147.18/3

```
procedure Assign (Target : in out Vector; Source : in Vector)             147.19/3
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target) and then
                Capacity (Target) >= Length (Target);
```

If Target denotes the same object as Source, the operation has no effect. If the length of Source is greater than the capacity of Target, Reserve_Capacity (Target, Length (Source)) is called. The elements of Source are then copied to Target as for an assignment_statement assigning Source to Target (this includes setting the length of Target to be that of Source). 147.20/3

```
function Copy (Source : Vector; Capacity : Count_Type := 0)               147.21/5
   return Vector
   with Pre => Capacity = 0 or else Capacity >= Length (Source)
                    or else raise Capacity_Error,
        Post => Length (Copy'Result) = Length (Source) and then
                not Tampering_With_Elements_Prohibited (Copy'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Copy'Result)
                   and then
                Copy'Result.Capacity >= (if Capacity = 0 then
                   Length (Source) else Capacity);
```

Returns a vector whose elements are initialized from the corresponding elements of Source. 147.22/5

```
procedure Move (Target : in out Vector;                                   148/5
                Source : in out Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Length (Target) = Length (Source)'Old and then
                    Length (Source) = 0 and then
                    Capacity (Target) >= Length (Source)'Old);
```

If Target denotes the same object as Source, then the operation has no effect. Otherwise, Move first calls Reserve_Capacity (Target, Length (Source)) and then Clear (Target); then, each element from Source is removed from Source and inserted into Target in the original order. 149/5

150/5
```
procedure Insert_Vector (Container : in out Vector;
                         Before    : in     Extended_Index;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);
```

151/5 If Length(New_Item) is 0, then Insert_Vector does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Length (New_Item); if the value of Last appropriate for length *NL* would be greater than Index_Type'Last, then Constraint_Error is propagated.

152/5 If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert_Vector slides the elements in the range Before .. Last_Index (Container) up by Length(New_Item) positions, and then copies the elements of New_Item to the positions starting at Before. Any exception raised during the copying is propagated.

153/5
```
procedure Insert_Vector (Container : in out Vector;
                         Before    : in     Cursor;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Capacity (Container) >= Length (Container);
```

154/5 If Length(New_Item) is 0, then Insert_Vector does nothing. If Before is No_Element, then the call is equivalent to Insert_Vector (Container, Last_Index (Container) + 1, New_Item); otherwise, the call is equivalent to Insert_Vector (Container, To_Index (Before), New_Item);

155/5
```
procedure Insert_Vector (Container : in out Vector;
                         Before    : in     Cursor;
                         New_Item  : in     Vector;
                         Position  :    out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                   Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);
```

156/5 If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert_Vector (Container, *T*, New_Item) is called, and then Position is set to To_Cursor (Container, *T*).

```
procedure Insert (Container : in out Vector;                                    157/5
                  Before    : in     Extended_Index;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

Equivalent to Insert (Container, Before, To_Vector (New_Item, Count)); 158/2

```
procedure Insert (Container : in out Vector;                                    159/5
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

Equivalent to Insert (Container, Before, To_Vector (New_Item, Count)); 160/2

```
procedure Insert (Container : in out Vector;                                    161/5
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);
```

Equivalent to Insert (Container, Before, To_Vector (New_Item, Count), Position); 162/2

```
procedure Insert (Container : in out Vector;                                    163/5
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

If Count is 0, then Insert does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last, then Constraint_Error is propagated. 164/5

165/2 If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts elements that are initialized by default (see 3.3.1) in the positions starting at Before.

166/5
```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);
```

167/5 If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*).

168/5
```
procedure Prepend_Vector (Container : in out Vector;
                          New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                    Length (Container) and then
                Capacity (Container) >= Length (Container);
```

169/2 Equivalent to Insert (Container, First_Index (Container), New_Item).

170/5
```
procedure Prepend (Container : in out Vector;
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

171/2 Equivalent to Insert (Container, First_Index (Container), New_Item, Count).

172/5
```
procedure Append_Vector (Container : in out Vector;
                         New_Item  : in     Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Length (New_Item)
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Length (New_Item) =
                    Length (Container) and then
                Capacity (Container) >= Length (Container);
```

173/2 Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item).

```
procedure Append (Container : in out Vector;                                    174/5
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item, Count). 175/2

```
procedure Append (Container : in out Vector;                                    175.1/5
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - 1
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + 1 = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

Equivalent to Insert (Container, Last_Index (Container) + 1, New_Item, 1). 175.2/5

```
procedure Insert_Space (Container : in out Vector;                              176/5
                        Before    : in     Extended_Index;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before in
                   First_Index (Container) .. Last_Index (Container) + 1
                   or else raise Constraint_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Capacity (Container) >= Length (Container);
```

If Count is 0, then Insert_Space does nothing. Otherwise, it computes the new length *NL* as the sum of the current length and Count; if the value of Last appropriate for length *NL* would be greater than Index_Type'Last, then Constraint_Error is propagated. 177/5

If the current vector capacity is less than *NL*, Reserve_Capacity (Container, *NL*) is called to increase the vector capacity. Then Insert_Space slides the elements in the range Before .. Last_Index (Container) up by Count positions, and then inserts empty elements in the positions starting at Before. 178/2

```
procedure Insert_Space (Container : in out Vector;                              179/5
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                   Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Maximum_Length - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container) and then
                Has_Element (Container, Position) and then
                Capacity (Container) >= Length (Container);
```

If Before equals No_Element, then let *T* be Last_Index (Container) + 1; otherwise, let *T* be To_Index (Before). Insert_Space (Container, *T*, Count) is called, and then Position is set to To_Cursor (Container, *T*). 180/5

181/5
```
procedure Delete (Container : in out Vector;
                  Index     : in     Extended_Index;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Index in
                    First_Index (Container) .. Last_Index (Container) + 1
                    or else raise Constraint_Error),
        Post => Length (Container)'Old - Count <= Length (Container);
```

182/5 If Count is 0, Delete has no effect. Otherwise, Delete slides the elements (if any) starting at position Index + Count down to Index. Any exception raised during element assignment is propagated.

183/5
```
procedure Delete (Container : in out Vector;
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Length (Container)'Old - Count <= Length (Container)
                and then Position = No_Element;
```

184/5 Delete (Container, To_Index (Position), Count) is called, and then Position is set to No_Element.

185/5
```
procedure Delete_First (Container : in out Vector;
                        Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

186/2 Equivalent to Delete (Container, First_Index (Container), Count).

187/5
```
procedure Delete_Last (Container : in out Vector;
                       Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

188/3 If Length (Container) <= Count, then Delete_Last is equivalent to Clear (Container). Otherwise, it is equivalent to Delete (Container, Index_Type'Val(Index_Type'Pos(Last_Index (Container)) – Count + 1), Count).

188.1/6
```
function Slice (Container : Vector;
                Low       : Index_Type;
                High      : Extended_Index) return Vector
   with Pre  => High in Low - 1 .. Container.Last_Index
                    or else raise Constraint_Error,
        Post => Slice'Result.Length = High - Low + 1 and then
                not Tampering_With_Elements_Prohibited (Slice'Result)
                   and then
                not Tampering_With_Cursors_Prohibited (Slice'Result)
                   and then
                Slice'Result.Capacity >= Slice'Result.Length;
```

188.2/6 This returns a slice of Container from Low to High. If High = Low - 1, then the result is an empty vector.

```
procedure Slice (Container : in out Vector;                                    188.3/6
                 Low       : in     Index_Type;
                 High      : in     Extended_Index)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (High in Low - 1 .. Container.Last_Index
                    or else raise Constraint_Error),
        Post => Container.Length = High - Low + 1 and then
                Container.Capacity >= Container.Length;
```

Equivalent to Container := Slice (Container, Low, High); 188.4/6

```
function Replace_Slice (Container : Vector;                                    188.5/6
                        Low       : Index_Type;
                        High      : Extended_Index;
                        By        : Vector) return Vector
   with Pre  => (High in Low - 1 .. Container.Last_Index
                    or else raise Constraint_Error) and then
                 (Container.Length - Count_Type (High - Low + 1) <=
                    Maximum_Length - By.Length
                    or else raise Constraint_Error),
        Post => Replace_Slice'Result.Length = Container.Length -
                    Count_Type (High - Low + 1) + By.Length and then
                 not Tampering_With_Elements_Prohibited
                    (Replace_Slice'Result) and then
                 not Tampering_With_Cursors_Prohibited
                    (Replace_Slice'Result) and then
                 Replace_Slice'Result.Capacity >= Replace_Slice.Length;
```

This returns a copy of Container with the slice of elements from Low to High replaced with the vector By. 188.6/6

```
procedure Replace_Slice (Container : in out Vector;                            188.7/6
                         Low       : in     Index_Type;
                         High      : in     Extended_Index;
                         By        : in     Vector)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                 (High in Low - 1 .. Container.Last_Index
                    or else raise Constraint_Error) and then
                 (Container.Length - Count_Type (High - Low + 1) <=
                    Maximum_Length - By.Length
                    or else raise Constraint_Error),
        Post => Container.Length = Container.Length'Old -
                    Count_Type (High - Low + 1) + By.Length and then
                 Container.Capacity >= Container.Length;
```

Equivalent to Container := Replace_Slice (Container, Low, High, By); If High = Low - 1, then this is equivalent to Insert_Vector (Container, Low, By); 188.8/6

```
procedure Reverse_Elements (Container : in out Vector)                         189/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error;
```

Reorders the elements of Container in reverse order. 190/2

```
procedure Swap (Container : in out Vector;                                     191/5
                I, J      : in     Index_Type)
   with Pre => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
               (I in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error) and then
               (J in First_Index (Container) .. Last_Index (Container)
                   or else raise Constraint_Error);
```

Swap exchanges the values of the elements at positions I and J. 192/5

193/5
```
procedure Swap (Container : in out Vector;
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                   or else raise Program_Error) and then
                (Has_Element (Container, J)
                   or else raise Program_Error);
```

194/5 Swap exchanges the values of the elements designated by I and J.

195/5
```
function First_Index (Container : Vector) return Index_Type
   with Nonblocking, Global => null, Use_Formal => null,
        Post => First_Index'Result = Index_Type'First;
```

196/2 Returns the value Index_Type'First.

197/5
```
function First (Container : Vector) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

198/2 If Container is empty, First returns No_Element. Otherwise, it returns a cursor that designates the first element in Container.

199/5
```
function First_Element (Container : Vector)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

200/2 Equivalent to Element (Container, First_Index (Container)).

201/5
```
function Last_Index (Container : Vector) return Extended_Index
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if Length (Container) = 0 then Last_Index'Result = No_Index
                 else Count_Type(Last_Index'Result - Index_Type'First) =
                      Length (Container) - 1);
```

202/2 If Container is empty, Last_Index returns No_Index. Otherwise, it returns the position of the last element in Container.

203/5
```
function Last (Container : Vector) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

204/2 If Container is empty, Last returns No_Element. Otherwise, it returns a cursor that designates the last element in Container.

205/5
```
function Last_Element (Container : Vector)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

206/2 Equivalent to Element (Container, Last_Index (Container)).

```
function Next (Position : Cursor) return Cursor                                    207/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) + 1 in the same vector as Position. 208/2

```
function Next (Container : Vector;                                                 208.1/5
               Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Has_Element (Container, Next'Result) then
                    To_Index (Container, Next'Result) =
                    To_Index (Container, Position) + 1
                 elsif Next'Result = No_Element then
                    Position = Last (Container)
                 else False);
```

Returns a cursor designating the next element in Container, if any. 208.2/5

```
procedure Next (Position : in out Cursor)                                          209/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

Equivalent to Position := Next (Position). 210/2

```
procedure Next (Container : in     Vector;                                         210.1/5
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

Equivalent to Position := Next (Container, Position). 210.2/5

```
function Previous (Position : Cursor) return Cursor                                211/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                 then Previous'Result = No_Element);
```

If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the element with index To_Index (Position) – 1 in the same vector as Position. 212/2

```
function Previous (Container : Vector;                                             212.1/5
                   Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Previous'Result = No_Element
                 elsif Has_Element (Container, Previous'Result) then
                   To_Index (Container, Previous'Result) =
                   To_Index (Container, Position) - 1
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else False);
```

Returns a cursor designating the previous element in Container, if any. 212.2/5

213/5
```
procedure Previous (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

214/2 Equivalent to Position := Previous (Position).

214.1/5
```
procedure Previous (Container : in     Vector;
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

214.2/5 Equivalent to Position := Previous (Container, Position).

215/2
```
function Find_Index (Container : Vector;
                     Item      : Element_Type;
                     Index     : Index_Type := Index_Type'First)
   return Extended_Index;
```

216/2 Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index and proceeds towards Last_Index (Container). If no equal element is found, then Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered.

217/5
```
function Find (Container : Vector;
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));
```

218/5 Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the first element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the last element of Container. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

219/2
```
function Reverse_Find_Index (Container : Vector;
                             Item      : Element_Type;
                             Index     : Index_Type := Index_Type'Last)
   return Extended_Index;
```

220/2 Searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at position Index or, if Index is greater than Last_Index (Container), at position Last_Index (Container). It proceeds towards First_Index (Container). If no equal element is found, then Reverse_Find_Index returns No_Index. Otherwise, it returns the index of the first equal element encountered.

```
function Reverse_Find (Container : Vector;                                    221/5
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Reverse_Find'Result /= No_Element
                 then Has_Element (Container, Reverse_Find'Result));
```

Reverse_Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the last element if Position equals No_Element, and at the element designated by Position otherwise. It proceeds towards the first element of Container. If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 222/5

```
function Contains (Container : Vector;                                        223/2
                   Item      : Element_Type) return Boolean;
```

Equivalent to Has_Element (Find (Container, Item)). 224/2

Paragraphs 225 and 226 were moved above.

```
procedure Iterate                                                             227/5
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

Invokes Process.**all** with a cursor that designates each element in Container, in index order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 228/3

```
procedure Reverse_Iterate                                                     229/5
  (Container : in Vector;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

Iterates over the elements in Container as per procedure Iterate, except that elements are traversed in reverse index order. 230/3

```
function Iterate (Container : in Vector)                                      230.1/5
   return Vector_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post   => Tampering_With_Cursors_Prohibited (Container);
```

Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor as per the Next function when used as a forward iterator, and starting with the last node and moving the cursor as per the Previous function when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 230.2/5

230.3/5
```
function Iterate (Container : in Vector; Start : in Cursor)
   return Vector_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre    => (Start /= No_Element
                      or else raise Constraint_Error) and then
                  (Has_Element (Container, Start)
                      or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container);
```

230.4/5 Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the node designated by Start and moving the cursor as per the Next function when used as a forward iterator, or moving the cursor as per the Previous function when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

231/3 The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. The number of times the subprograms of Generic_Sorting call "<" is unspecified.

232/2
```
function Is_Sorted (Container : Vector) return Boolean;
```

233/2 Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated.

234/5
```
procedure Sort (Container : in out Vector)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;
```

235/2 Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

236/5
```
procedure Merge (Target  : in out Vector;
                 Source  : in out Vector)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error) and then
                (Length (Target) <= Maximum_Length - Length (Source)
                    or else raise Constraint_Error) and then
                ((Length (Source) = 0 or else
                  not Target'Has_Same_Storage (Source))
                  or else raise Program_Error),
        Post => (declare
                    Result_Length : constant Count_Type :=
                       Length (Source)'Old + Length (Target)'Old;
                 begin
                    (Length (Source) = 0 and then
                     Length (Target) = Result_Length and then
                     Capacity (Target) >= Result_Length));
```

237/5 Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated.

The nested package Vectors.Stable provides a type Stable.Vector that represents a *stable* vector, which is one that cannot grow and shrink. Such a vector can be created by calling the To_Vector or Copy functions, or by establishing a *stabilized view* of an ordinary vector. 237.1/5

The subprograms of package Containers.Vectors that have a parameter or result of type Vector are included in the nested package Stable with the same specification, except that the following are omitted: 237.2/5

Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Reserve_Capacity, Assign, Move, Insert, Insert_Space, Insert_Vector, Append, Append_Vector, Prepend, Prepend_Vector, Clear, Delete, Delete_First, Delete_Last, and Set_Length 237.3/5

The generic package Generic_Sorting is also included with the same specification, except that Merge is omitted. 237.4/5

The operations of this package are equivalent to those for ordinary vectors, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 237.5/5

If a stable vector is declared with the Base discriminant designating a pre-existing ordinary vector, the stable vector represents a stabilized view of the underlying ordinary vector, and any operation on the stable vector is reflected on the underlying ordinary vector. While a stabilized view exists, any operation that tampers with elements performed on the underlying vector is prohibited. The finalization of a stable vector that provides such a view removes this restriction on the underlying ordinary vector (though some other restriction can exist due to other concurrent iterations or stabilized views). 237.6/5

If a stable vector is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable vector, typically a call on To_Vector or Copy, determines the Length of the vector. The Length of a stable vector never changes after initialization. 237.7/5

Bounded (Run-Time) Errors

Reading the value of an empty element by calling Element, Query_Element, Update_Element, Constant_Reference, Reference, Swap, Is_Sorted, Sort, Merge, "=", Find, or Reverse_Find is a bounded error. The implementation may treat the element as having any normal value (see 13.9.1) of the element type, or raise Constraint_Error or Program_Error before modifying the vector. 238/3

Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined. 239/2

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any Vector parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the Vector either prior to, or subsequent to, some or all of the modifications to the Vector. 239.1/3

It is a bounded error to call any subprogram declared in the visible part of Containers.Vectors when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error. 239.2/3

A Cursor value is *ambiguous* if any of the following have occurred since it was created: 240/2

- Insert, Insert_Space, Insert_Vector, or Delete has been called on the vector that contains the element the cursor designates with an index value (or a cursor designating an element at such an index value) less than or equal to the index value of the element designated by the cursor; or 241/5

242/2 • The vector that contains the element it designates has been passed to the Sort or Merge procedures of an instance of Generic_Sorting, or to the Reverse_Elements procedure.

243/2 It is a bounded error to call any subprogram other than "=" or Has_Element declared in Containers.Vectors with an ambiguous (but not invalid, see below) cursor parameter. Possible results are:

244/2 • The cursor may be treated as if it were No_Element;

245/2 • The cursor may designate some element in the vector (but not necessarily the element that it originally designated);

246/2 • Constraint_Error may be raised; or

247/2 • Program_Error may be raised.

Erroneous Execution

248/2 A Cursor value is *invalid* if any of the following have occurred since it was created:

249/2 • The vector that contains the element it designates has been finalized;

249.1/3 • The vector that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

250/2 • The vector that contains the element it designates has been used as the Source or Target of a call to Move; or

251/3 • The element it designates has been deleted or removed from the vector that previously contained the element.

252/2 The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Vectors is called with an invalid cursor parameter.

252.1/3 Execution is erroneous if the vector associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

Implementation Requirements

253/2 No storage associated with a vector object shall be lost upon assignment or scope exit.

254/3 The execution of an assignment_statement for a vector shall have the effect of copying the elements from the source vector object to the target vector object and changing the length of the target object to that of the source object.

Implementation Advice

255/2 Containers.Vectors should be implemented similarly to an array. In particular, if the length of a vector is *N*, then

256/2 • the worst-case time complexity of Element should be $O(\log N)$;

257/2 • the worst-case time complexity of Append with Count=1 when *N* is less than the capacity of the vector should be $O(\log N)$; and

258/2 • the worst-case time complexity of Prepend with Count=1 and Delete_First with Count=1 should be $O(N \log N)$.

259/2 The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be $O(N^{**}2)$, and the average time complexity should be better than $O(N^{**}2)$.

Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements. 260/2

Move should not copy elements, and should minimize copying of internal data structures. 261/2

If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation. 262/2

NOTE 1 All elements of a vector occupy locations in the internal array. If a sparse container is required, a Hashed_Map can be used rather than a vector. 263/5

NOTE 2 If Index_Type'Base'First = Index_Type'First an instance of Ada.Containers.Vectors will raise Constraint_Error. A value below Index_Type'First is required so that an empty vector has a meaningful value of Last_Index. 264/2

A.18.3 The Generic Package Containers.Doubly_Linked_Lists

The language-defined generic package Containers.Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. A list container is optimized for insertion and deletion at any position. 1/2

A doubly-linked list container object manages a linked list of internal *nodes*, each of which contains an element and pointers to the next (successor) and previous (predecessor) internal nodes. A cursor designates a particular node within a list (and by extension the element contained in that node). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved in the container. 2/2

The *length* of a list is the number of elements it contains. 3/2

Static Semantics

The generic library package Containers.Doubly_Linked_Lists has the following declaration: 4/2

```
with Ada.Iterator_Interfaces;                                                    5/5
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Doubly_Linked_Lists
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is

   type List is tagged private                                                   6/5
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.List,
           Aggregate         => (Empty       => Empty,
                                 Add_Unnamed => Append),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (List) = 0 and then
              (not Tampering_With_Cursors_Prohibited (List)) and then
              (not Tampering_With_Elements_Prohibited (List)),
           Preelaborable_Initialization;

   type Cursor is private                                                        7/5
      with Preelaborable_Initialization;

   Empty_List : constant List;                                                   8/2

   No_Element : constant Cursor;                                                 9/2
```

```
9.1/5       function Has_Element (Position : Cursor) return Boolean
               with Nonblocking, Global => in all, Use_Formal => null;

9.2/5       function Has_Element (Container : List; Position : Cursor)
               return Boolean
               with Nonblocking, Global => null, Use_Formal => null;

9.3/3       package List_Iterator_Interfaces is new
                Ada.Iterator_Interfaces (Cursor, Has_Element);

10/2        function "=" (Left, Right : List) return Boolean;

10.1/5      function Tampering_With_Cursors_Prohibited
               (Container : List) return Boolean
               with Nonblocking, Global => null, Use_Formal => null;

10.2/5      function Tampering_With_Elements_Prohibited
               (Container : List) return Boolean
               with Nonblocking, Global => null, Use_Formal => null;

10.3/5      function Empty return List
               is (Empty_List)
               with Post =>
                      not Tampering_With_Elements_Prohibited (Empty'Result) and then
                      not Tampering_With_Cursors_Prohibited (Empty'Result) and then
                      Length (Empty'Result) = 0;

11/5        function Length (Container : List) return Count_Type
               with Nonblocking, Global => null, Use_Formal => null;

12/5        function Is_Empty (Container : List) return Boolean
               with Nonblocking, Global => null, Use_Formal => null,
                    Post => Is_Empty'Result = (Length (Container) = 0);

13/5        procedure Clear (Container : in out List)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                               or else raise Program_Error,
                    Post => Length (Container) = 0;

14/5        function Element (Position : Cursor) return Element_Type
               with Pre  => Position /= No_Element or else raise Constraint_Error,
                    Nonblocking, Global => in all, Use_Formal => Element_Type;

14.1/5      function Element (Container : List;
                              Position  : Cursor) return Element_Type
               with Pre => (Position /= No_Element or else
                               raise Constraint_Error) and then
                           (Has_Element (Container, Position)
                               or else raise Program_Error),
                    Nonblocking, Global => null, Use_Formal => Element_Type;

15/5        procedure Replace_Element (Container : in out List;
                                       Position  : in     Cursor;
                                       New_item  : in     Element_Type)
               with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                               or else raise Program_Error) and then
                            (Position /= No_Element
                               or else raise Constraint_Error) and then
                            (Has_Element (Container, Position)
                               or else raise Program_Error);

16/5        procedure Query_Element
              (Position : in Cursor;
               Process  : not null access procedure (Element : in Element_Type))
               with Pre  => Position /= No_Element or else raise Constraint_Error,
                    Global => in all;
```

```
procedure Query_Element                                                    16.1/5
  (Container : in List;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);

procedure Update_Element                                                   17/5
  (Container : in out List;
   Position  : in     Cursor;
   Process   : not null access procedure
                  (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);

type Constant_Reference_Type                                               17.1/6
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);

type Reference_Type                                                        17.2/6
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);

function Constant_Reference (Container : aliased in List;                  17.3/5
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element or else
                    raise Constraint_Error) and then
                (Has_Element (Container, Position) or else
                    raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;

function Reference (Container : aliased in out List;                       17.4/5
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element or else
                    raise Constraint_Error) and then
                (Has_Element (Container, Position) or else
                    raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;

procedure Assign (Target : in out List; Source : in List)                  17.5/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Source) = Length (Target);

function Copy (Source : List)                                              17.6/5
   return List
   with Post =>
          Length (Copy'Result) = Length (Source) and then
          not Tampering_With_Elements_Prohibited (Copy'Result) and then
          not Tampering_With_Cursors_Prohibited (Copy'Result);
```

18/5
```
      procedure Move (Target : in out List;
                      Source : in out List)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                         or else raise Program_Error) and then
                      (not Tampering_With_Cursors_Prohibited (Source)
                         or else raise Program_Error),
              Post => (if not Target'Has_Same_Storage (Source) then
                         Length (Target) = Length (Source'Old) and then
                         Length (Source) = 0);
```

19/5
```
      procedure Insert (Container : in out List;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Has_Element (Container, Before)
                         or else raise Program_Error) and then
                      (Length (Container) <= Count_Type'Last - Count
                         or else raise Constraint_Error),
              Post => Length (Container)'Old + Count = Length (Container);
```

20/5
```
      procedure Insert (Container : in out List;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Has_Element (Container, Before)
                         or else raise Program_Error) and then
                      (Length (Container) <= Count_Type'Last - Count
                         or else raise Constraint_Error),
              Post => Length (Container)'Old + Count = Length (Container)
                      and then Has_Element (Container, Position);
```

21/5
```
      procedure Insert (Container : in out List;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Has_Element (Container, Before)
                         or else raise Program_Error) and then
                      (Length (Container) <= Count_Type'Last - Count
                         or else raise Constraint_Error),
              Post => Length (Container)'Old + Count = Length (Container)
                      and then Has_Element (Container, Position);
```

22/5
```
      procedure Prepend (Container : in out List;
                         New_Item  : in     Element_Type;
                         Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                          or else raise Program_Error) and then
                       (Length (Container) <= Count_Type'Last - Count
                          or else raise Constraint_Error),
              Post => Length (Container)'Old + Count = Length (Container);
```

23/5
```
      procedure Append (Container : in out List;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                          or else raise Program_Error) and then
                      (Length (Container) <= Count_Type'Last - Count
                         or else raise Constraint_Error),
              Post => Length (Container)'Old + Count = Length (Container);
```

```
procedure Append (Container : in out List;                                    23.1/5
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => Length (Container)'Old + 1 = Length (Container);

procedure Delete (Container : in out List;                                    24/5
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Length (Container)'Old - Count <= Length (Container)
                and then Position = No_Element;

procedure Delete_First (Container : in out List;                              25/5
                        Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);

procedure Delete_Last (Container : in out List;                               26/5
                       Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);

procedure Reverse_Elements (Container : in out List)                          27/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;

procedure Swap (Container : in out List;                                      28/5
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                    or else raise Program_Error) and then
                (Has_Element (Container, J)
                    or else raise Program_Error);

procedure Swap_Links (Container : in out List;                                29/5
                      I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                    or else raise Program_Error) and then
                (Has_Element (Container, J)
                    or else raise Program_Error);
```

30/5
```
      procedure Splice (Target   : in out List;
                        Before   : in     Cursor;
                        Source   : in out List)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                         or else raise Program_Error) and then
                      (not Tampering_With_Cursors_Prohibited (Source)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Has_Element (Target, Before)
                         or else raise Program_Error) and then
                      (Target'Has_Same_Storage (Source) or else
                       Length (Target) <= Count_Type'Last - Length (Source)
                         or else raise Constraint_Error),
              Post => (if not Target'Has_Same_Storage (Source) then
                         (declare
                            Result_Length : constant Count_Type :=
                               Length (Source)'Old + Length (Target)'Old;
                          begin
                            Length (Source) = 0 and then
                            Length (Target) = Result_Length));
```

31/5
```
      procedure Splice (Target   : in out List;
                        Before   : in     Cursor;
                        Source   : in out List;
                        Position : in out Cursor)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                         or else raise Program_Error) and then
                      (not Tampering_With_Cursors_Prohibited (Source)
                         or else raise Program_Error) and then
                      (Position /= No_Element
                         or else raise Constraint_Error) and then
                      (Has_Element (Source, Position)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Has_Element (Target, Before)
                         or else raise Program_Error) and then
                      (Target'Has_Same_Storage (Source) or else
                       Length (Target) <= Count_Type'Last - 1
                         or else raise Constraint_Error),
              Post => (declare
                         Org_Target_Length : constant Count_Type :=
                            Length (Target)'Old;
                         Org_Source_Length : constant Count_Type :=
                            Length (Source)'Old;
                       begin
                          (if Target'Has_Same_Storage (Source) then
                              Position = Position'Old
                           else
                              Length (Source) = Org_Source_Length - 1 and then
                              Length (Target) = Org_Target_Length + 1 and then
                              Has_Element (Target, Position)));
```

32/2
```
      procedure Splice (Container: in out List;
                        Before   : in     Cursor;
                        Position : in     Cursor)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Position /= No_Element
                         or else raise Constraint_Error) and then
                      (Has_Element (Container, Position)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Has_Element (Container, Before)
                         or else raise Program_Error),
              Post =>  Length (Container) = Length (Container)'Old;
```

```
function First (Container : List) return Cursor                                    33/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);

function First_Element (Container : List)                                          34/5
   return Element_Type
   with Pre => (not Is_Empty (Container)
                  or else raise Constraint_Error);

function Last (Container : List) return Cursor                                     35/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);

function Last_Element (Container : List)                                           36/5
   return Element_Type
   with Pre => (not Is_Empty (Container)
                  or else raise Constraint_Error);

function Next (Position : Cursor) return Cursor                                    37/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);

function Next (Container : List;                                                   37.1/5
               Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                  or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));

function Previous (Position : Cursor) return Cursor                                38/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element);

function Previous (Container : List;                                               38.1/5
                   Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                  or else raise Program_Error,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else Has_Element (Container, Previous'Result));

procedure Next (Position : in out Cursor)                                          39/5
   with Nonblocking, Global => in all, Use_Formal => null;

procedure Next (Container : in     List;                                           39.1/5
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                  or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));

procedure Previous (Position : in out Cursor)                                      40/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

```
40.1/5          procedure Previous (Container : in     List;
                                    Position  : in out Cursor)
                   with Nonblocking, Global => null, Use_Formal => null,
                        Pre  => Position = No_Element or else
                                Has_Element (Container, Position)
                                   or else raise Program_Error,
                        Post => (if Position /= No_Element then
                                   Has_Element (Container, Position));

41/5            function Find (Container : List;
                               Item      : Element_Type;
                               Position  : Cursor := No_Element)
                   return Cursor
                   with Pre  => Position = No_Element or else
                                Has_Element (Container, Position)
                                   or else raise Program_Error,
                        Post => (if Find'Result /= No_Element
                                 then Has_Element (Container, Find'Result));

42/5            function Reverse_Find (Container : List;
                                       Item      : Element_Type;
                                       Position  : Cursor := No_Element)
                   return Cursor
                   with Pre  => Position = No_Element or else
                                Has_Element (Container, Position)
                                   or else raise Program_Error,
                        Post => (if Reverse_Find'Result /= No_Element
                                 then Has_Element (Container, Reverse_Find'Result));

43/2            function Contains (Container : List;
                                   Item      : Element_Type) return Boolean;

44/3        This paragraph was deleted.

45/5            procedure  Iterate
                  (Container : in List;
                   Process   : not null access procedure (Position : in Cursor))
                   with Allows_Exit;

46/5            procedure Reverse_Iterate
                  (Container : in List;
                   Process   : not null access procedure (Position : in Cursor))
                   with Allows_Exit;

46.1/5          function Iterate (Container : in List)
                   return List_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
                   with Post   => Tampering_With_Cursors_Prohibited (Container);

46.2/5          function Iterate (Container : in List; Start : in Cursor)
                   return List_Iterator_Interfaces.Reversible_Iterator'Class
                   with Pre    => (Start /= No_Element
                                      or else raise Constraint_Error) and then
                                  (Has_Element (Container, Start)
                                      or else raise Program_Error),
                        Post   => Tampering_With_Cursors_Prohibited (Container);

47/5            generic
                   with function "<" (Left, Right : Element_Type)
                      return Boolean is <>;
                package Generic_Sorting
                with Nonblocking, Global => null is

48/2               function Is_Sorted (Container : List) return Boolean;

49/5               procedure Sort (Container : in out List)
                      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                      or else raise Program_Error;
```

```
   procedure Merge (Target  : in out List;                                    50/5
                    Source  : in out List)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error) and then
                   (not Tampering_With_Elements_Prohibited (Source)
                      or else raise Program_Error) and then
                   (Length (Target) <= Count_Type'Last - Length (Source)
                      or else raise Constraint_Error) and then
                   ((Length (Source) = 0 or else
                      not Target'Has_Same_Storage (Source))
                      or else raise Constraint_Error),
           Post => (declare
                      Result_Length : constant Count_Type :=
                         Length (Source)'Old + Length (Target)'Old;
                    begin
                      (Length (Source) = 0 and then
                       Length (Target) = Result_Length));
end Generic_Sorting;                                                          51/2

package Stable is                                                             51.1/5

   type List (Base : not null access Doubly_Linked_Lists.List) is             51.2/5
      tagged limited private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Stable_Properties => (Length),
           Global => null,
           Default_Initial_Condition => Length (List) = 0,
           Preelaborable_Initialization;

   type Cursor is private                                                     51.3/5
      with Preelaborable_Initialization;

   Empty_List : constant List;                                               51.4/5

   No_Element : constant Cursor;                                             51.5/5

   function Has_Element (Position : Cursor) return Boolean                   51.6/5
      with Nonblocking, Global => in all, Use_Formal => null;

   package List_Iterator_Interfaces is new                                   51.7/5
      Ada.Iterator_Interfaces (Cursor, Has_Element);

   procedure Assign (Target : in out Doubly_Linked_Lists.List;               51.8/5
                     Source : in List)
      with Post => Length (Source) = Length (Target);

   function Copy (Source : Doubly_Linked_Lists.List) return List             51.9/5
      with Post => Length (Copy'Result) = Length (Source);

   type Constant_Reference_Type                                              51.10/6
         (Element : not null access constant Element_Type) is
            limited private
      with Implicit_Dereference => Element,
           Nonblocking, Global => null, Use_Formal => null,
           Default_Initial_Condition => (raise Program_Error);

   type Reference_Type                                                       51.11/6
         (Element : not null access Element_Type) is limited private
      with Implicit_Dereference => Element,
           Nonblocking, Global => null, Use_Formal => null,
           Default_Initial_Condition => (raise Program_Error);

   -- Additional subprograms as described in the text                        51.12/5
   -- are declared here.

private                                                                      51.13/5

   ... -- not specified by the language                                      51.14/5

end Stable;                                                                  51.15/5
```

52/2
```
private
```
53/2
```
   ... -- not specified by the language
```
54/2
```
end Ada.Containers.Doubly_Linked_Lists;
```

55/2 The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions Find, Reverse_Find, and "=" on list values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions Find, Reverse_Find, and "=" on list values are unspecified.

56/2 The type List is used to represent lists. The type List needs finalization (see 7.6).

57/2 Empty_List represents the empty List object. It has a length of 0. If an object of type List is not otherwise initialized, it is initialized to the same value as Empty_List.

58/2 No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

59/5 The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

60/2 Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

60.1/5 List'Write for a List object *L* writes Length(*L*) elements of the list to the stream. It may also write additional information about the list.

60.2/3 List'Read reads the representation of a list from the stream, and assigns to *Item* a list with the same length and elements as was written by List'Write.

61/5 Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced. When tampering with cursors is *prohibited* for a particular list object *L*, Program_Error is propagated by the finalization of *L*, as well as by a call that passes *L* to certain of the operations of this package, as indicated by the precondition of such an operation. Similarly, when tampering with elements is *prohibited* for *L*, Program_Error is propagated by a call that passes *L* to certain of the other operations of this package, as indicated by the precondition of such an operation.

Paragraphs 62 through 69 are removed as preconditions now describe these rules.

69.2/5
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```
69.3/3 Returns True if Position designates an element, and returns False otherwise.

69.4/5
```
function Has_Element (Container : List; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
69.5/5 Returns True if Position designates an element in Container, and returns False otherwise.

70/2
```
function "=" (Left, Right : List) return Boolean;
```
71/3 If Left and Right denote the same list object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, it compares each element in Left to the corresponding element in Right using the generic formal equality operator. If any

such comparison returns False, the function returns False; otherwise, it returns True. Any exception raised during evaluation of element equality is propagated.

```
function Tampering_With_Cursors_Prohibited                                    71.1/5
   (Container : List) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise. 71.2/5

```
function Tampering_With_Elements_Prohibited                                   71.3/5
   (Container : List) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

Always returns False, regardless of whether tampering with elements is prohibited. 71.4/5

```
function Length (Container : List) return Count_Type                          72/5
   with Nonblocking, Global => null, Use_Formal => null;
```

Returns the number of elements in Container. 73/2

```
function Is_Empty (Container : List) return Boolean                           74/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```

Returns True if Container is empty. 75/5

```
procedure Clear (Container : in out List)                                     76/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

Removes all the elements from Container. 77/2

```
function Element (Position : Cursor) return Element_Type                      78/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

Element returns the element designated by Position. 79/5

```
function Element (Container : List;                                           79.1/5
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element or else
                   raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

Element returns the element designated by Position in Container. 79.2/5

```
procedure Replace_Element (Container : in out List;                           80/5
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

Replace_Element assigns the value New_Item to the element designated by Position. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object (including itself). 81/5

82/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

83/5 Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the list that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

83.1/5
```
procedure Query_Element
  (Container : in List;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

83.2/5 Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

84/5
```
procedure Update_Element
  (Container : in out List;
   Position  : in     Cursor;
   Process   : not null access procedure
                    (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

85/5 Update_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

86/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

86.1/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

86.2/6
```
type Reference_Type
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

86.3/3 The types Constant_Reference_Type and Reference_Type need finalization.

86.4/5 *This paragraph was deleted.*

```
function Constant_Reference (Container : aliased in List;                    86.5/5
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element or else
                    raise Constraint_Error) and then
                 (Has_Element (Container, Position) or else
                    raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a list given a cursor. 86.6/3

Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized. 86.7/5

```
function Reference (Container : aliased in out List;                         86.8/5
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element or else
                    raise Constraint_Error) and then
                 (Has_Element (Container, Position) or else
                    raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a list given a cursor. 86.9/3

Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized. 86.10/5

```
procedure Assign (Target : in out List; Source : in List)                    86.11/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Source) = Length (Target);
```

If Target denotes the same object as Source, the operation has no effect. Otherwise, the elements of Source are copied to Target as for an assignment_statement assigning Source to Target. 86.12/3

```
function Copy (Source : List)                                                86.13/5
   return List
   with Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result);
```

Returns a list whose elements match the elements of Source. 86.14/3

```
procedure Move (Target : in out List;                                        87/5
                Source : in out List)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Length (Target) = Length (Source'Old) and then
                    Length (Source) = 0);
```

If Target denotes the same object as Source, then the operation has no effect. Otherwise, the operation is equivalent to Assign (Target, Source) followed by Clear (Source). 88/3

89/5
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container);
```

90/5 Insert inserts Count copies of New_Item prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

91/5
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container)
                and then Has_Element (Container, Position);
```

92/5 Insert allocates Count copies of New_Item, and inserts them prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last element (if any). Position designates the first newly-inserted element, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

93/5
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container)
                and then Has_Element (Container, Position);
```

94/5 Insert inserts Count new elements prior to the element designated by Before. If Before equals No_Element, the new elements are inserted after the last node (if any). The new elements are initialized by default (see 3.3.1). Position designates the first newly-inserted element, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Prepend (Container : in out List;                                    95/5
                   New_Item  : in     Element_Type;
                   Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container);
```

Equivalent to Insert (Container, First (Container), New_Item, Count). 96/2

```
procedure Append (Container : in out List;                                     97/5
                  New_Item  : in     Element_Type;
                  Count     : in     Count_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - Count
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + Count = Length (Container);
```

Equivalent to Insert (Container, No_Element, New_Item, Count). 98/2

```
procedure Append (Container : in out List;                                     98.1/5
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => Length (Container)'Old + 1 = Length (Container);
```

Equivalent to Insert (Container, No_Element, New_Item, 1). 98.2/5

```
procedure Delete (Container : in out List;                                     99/5
                  Position  : in out Cursor;
                  Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Post => Length (Container)'Old - Count <= Length (Container)
                and then Position = No_Element;
```

Delete removes (from Container) Count elements starting at the element designated by Position (or all of the elements starting at Position if there are fewer than Count elements starting at Position). Finally, Position is set to No_Element. 100/5

```
procedure Delete_First (Container : in out List;                               101/5
                        Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

If Length (Container) <= Count, then Delete_First is equivalent to Clear (Container). Otherwise, it removes the first Count nodes from Container. 102/3

```
procedure Delete_Last (Container : in out List;                                103/5
                       Count     : in     Count_Type := 1)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Length (Container)'Old - Count <= Length (Container);
```

If Length (Container) <= Count, then Delete_Last is equivalent to Clear (Container). Otherwise, it removes the last Count nodes from Container. 104/3

105/5
```
procedure Reverse_Elements (Container : in out List)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;
```

106/2 Reorders the elements of Container in reverse order.

107/5
```
procedure Swap (Container : in out List;
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                    or else raise Program_Error) and then
                (Has_Element (Container, J)
                    or else raise Program_Error);
```

108/5 Swap exchanges the values of the elements designated by I and J.

109/5
```
procedure Swap_Links (Container : in out List;
                      I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                    or else raise Program_Error) and then
                (Has_Element (Container, J)
                    or else raise Program_Error);
```

110/5 Swap_Links exchanges the nodes designated by I and J.

111/5
```
procedure Splice (Target   : in out List;
                  Before   : in     Cursor;
                  Source   : in out List)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Target, Before)
                    or else raise Program_Error) and then
                (Target'Has_Same_Storage (Source) or else
                 Length (Target) <= Count_Type'Last - Length (Source)
                    or else raise Constraint_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    (declare
                       Result_Length : constant Count_Type :=
                           Length (Source)'Old + Length (Target)'Old;
                     begin
                       Length (Source) = 0 and then
                       Length (Target) = Result_Length));
```

112/5 If Source denotes the same object as Target, the operation has no effect. Otherwise, Splice reorders elements such that they are removed from Source and moved to Target, immediately prior to Before. If Before equals No_Element, the nodes of Source are spliced after the last node of Target.

```
procedure Splice (Target   : in out List;                                    113/5
                  Before   : in     Cursor;
                  Source   : in out List;
                  Position : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Source, Position)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Target, Before)
                   or else raise Program_Error) and then
                (Target'Has_Same_Storage (Source) or else
                 Length (Target) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Org_Target_Length : constant Count_Type :=
                      Length (Target)'Old;
                   Org_Source_Length : constant Count_Type :=
                      Length (Source)'Old;
                 begin
                    (if Target'Has_Same_Storage (Source) then
                        Position = Position'Old
                     else Length (Source) = Org_Source_Length - 1 and then
                        Length (Target) = Org_Target_Length + 1 and then
                        Has_Element (Target, Position)));
```

If Source denotes the same object as Target, then there is no effect if Position equals Before, else the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. Otherwise, the element designated by Position is removed from Source and moved to Target, immediately prior to Before, or, if Before equals No_Element, after the last element of Target. Position is updated to represent an element in Target. 114/5

```
procedure Splice (Container: in out List;                                    115/5
                  Before   : in     Cursor;
                  Position : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Has_Element (Container, Before)
                   or else raise Program_Error),
        Post =>  Length (Container) = Length (Container)'Old;
```

If Position equals Before there is no effect. Otherwise, the element designated by Position is moved immediately prior to Before, or, if Before equals No_Element, after the last element. 116/5

```
function First (Container : List) return Cursor                              117/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

If Container is empty, First returns No_Element. Otherwise, it returns a cursor that designates the first node in Container. 118/2

119/5
```
function First_Element (Container : List)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

120/2 Equivalent to Element (Container, First_Index (Container)).

121/5
```
function Last (Container : List) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

122/2 If Container is empty, Last returns No_Element. Otherwise, it returns a cursor that designates the last node in Container.

123/5
```
function Last_Element (Container : List)
   return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

124/2 Equivalent to Element (Last (Container)).

125/5
```
function Next (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

126/2 If Position equals No_Element or designates the last element of the container, then Next returns the value No_Element. Otherwise, it returns a cursor that designates the successor of the element designated by Position.

126.1/5
```
function Next (Container : List;
               Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));
```

126.2/5 Returns a cursor designating the successor of the element designated by Position in Container.

127/5
```
function Previous (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element);
```

128/2 If Position equals No_Element or designates the first element of the container, then Previous returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor of the element designated by Position.

```
function Previous (Container : List;                                    128.1/5
                   Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else Has_Element (Container, Previous'Result));
```

Returns a cursor designating the predecessor of the element designated by Position in Container, if any. 128.2/5

```
procedure Next (Position : in out Cursor)                               129/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

Equivalent to Position := Next (Position). 130/2

```
procedure Next (Container : in     List;                                130.1/5
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

Equivalent to Position := Next (Container, Position). 130.2/5

```
procedure Previous (Position : in out Cursor)                           131/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

Equivalent to Position := Previous (Position). 132/2

```
procedure Previous (Container : in     List;                            132.1/5
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

Equivalent to Position := Previous (Container, Position). 132.2/5

```
function Find (Container : List;                                        133/5
               Item      : Element_Type;
               Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));
```

Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the first element if Position equals No_Element. It proceeds towards Last (Container). If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 134/5

135/5
```
function Reverse_Find (Container : List;
                       Item      : Element_Type;
                       Position  : Cursor := No_Element)
   return Cursor
   with Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Reverse_Find'Result /= No_Element
                 then Has_Element (Container, Reverse_Find'Result));
```

136/5 Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position, or at the last element if Position equals No_Element. It proceeds towards First (Container). If no equal element is found, then Reverse_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

137/2
```
function Contains (Container : List;
                   Item      : Element_Type) return Boolean;
```

138/2 Equivalent to Find (Container, Item) /= No_Element.

Paragraphs 139 and 140 were moved above.

141/5
```
procedure Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

142/3 Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor as per the Next function. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

143/5
```
procedure Reverse_Iterate
  (Container : in List;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

144/3 Iterates over the nodes in Container as per procedure Iterate, except that elements are traversed in reverse order, starting with the last node and moving the cursor as per the Previous function.

144.1/5
```
function Iterate (Container : in List)
   return List_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post   => Tampering_With_Cursors_Prohibited (Container);
```

144.2/5 Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor as per the Next function when used as a forward iterator, and starting with the last node and moving the cursor as per the Previous function when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

```
function Iterate (Container : in List; Start : in Cursor)                    144.3/5
   return List_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre    => (Start /= No_Element
                      or else raise Constraint_Error) and then
                  (Has_Element (Container, Start)
                      or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container);
```

Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the node designated by Start and moving the cursor as per the Next function when used as a forward iterator, or moving the cursor as per the Previous function when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 144.4/5

The actual function for the generic formal function "<" of Generic_Sorting is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the subprograms of Generic_Sorting are unspecified. The number of times the subprograms of Generic_Sorting call "<" is unspecified. 145/3

```
function Is_Sorted (Container : List) return Boolean;                        146/2
```

Returns True if the elements are sorted smallest first as determined by the generic formal "<" operator; otherwise, Is_Sorted returns False. Any exception raised during evaluation of "<" is propagated. 147/2

```
procedure Sort (Container : in out List)                                     148/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error;
```

Reorders the nodes of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. The sort is stable. Any exception raised during evaluation of "<" is propagated. 149/2

```
procedure Merge (Target  : in out List;                                      150/5
                 Source  : in out List)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Elements_Prohibited (Source)
                    or else raise Program_Error) and then
                (Length (Target) <= Count_Type'Last - Length (Source)
                    or else raise Constraint_Error) and then
                ((Length (Source) = 0 or else
                    not Target'Has_Same_Storage (Source))
                    or else raise Constraint_Error),
        Post => (declare
                    Result_Length : constant Count_Type :=
                       Length (Source)'Old + Length (Target)'Old;
                 begin
                    (Length (Source) = 0 and then
                     Length (Target) = Result_Length));
```

Merge removes elements from Source and inserts them into Target; afterwards, Target contains the union of the elements that were initially in Source and Target; Source is left empty. If Target and Source are initially sorted smallest first, then Target is ordered smallest first as determined by the generic formal "<" operator; otherwise, the order of elements in Target is unspecified. Any exception raised during evaluation of "<" is propagated. 151/5

151.1/5 The nested package Doubly_Linked_Lists.Stable provides a type Stable.List that represents a *stable* list, which is one that cannot grow and shrink. Such a list can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary list.

151.2/5 The subprograms of package Containers.Doubly_Linked_Lists that have a parameter or result of type List are included in the nested package Stable with the same specification, except that the following are omitted:

151.3/5 Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Assign, Move, Insert, Append, Prepend, Clear, Delete, Delete_First, Delete_Last, Splice, Swap_Links, and Reverse_Elements

151.4/5 The operations of this package are equivalent to those for ordinary lists, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True.

151.5/5 If a stable list is declared with the Base discriminant designating a pre-existing ordinary list, the stable list represents a stabilized view of the underlying ordinary list, and any operation on the stable list is reflected on the underlying ordinary list. While a stabilized view exists, any operation that tampers with elements performed on the underlying list is prohibited. The finalization of a stable list that provides such a view removes this restriction on the underlying ordinary list (though some other restriction can exist due to other concurrent iterations or stabilized views).

151.6/5 If a stable list is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable list, typically a call on Copy, determines the Length of the list. The Length of a stable list never changes after initialization.

Bounded (Run-Time) Errors

152/2 Calling Merge in an instance of Generic_Sorting with either Source or Target not ordered smallest first using the provided generic formal "<" operator is a bounded error. Either Program_Error is raised after Target is updated as described for Merge, or the operation works as defined.

152.1/3 It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any List parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the List either prior to, or subsequent to, some or all of the modifications to the List.

152.2/3 It is a bounded error to call any subprogram declared in the visible part of Containers.Doubly_Linked_Lists when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

Erroneous Execution

153/2 A Cursor value is *invalid* if any of the following have occurred since it was created:

154/2 - The list that contains the element it designates has been finalized;

154.1/3 - The list that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

155/2 - The list that contains the element it designates has been used as the Source or Target of a call to Move; or

156/3 - The element it designates has been removed from the list that previously contained the element.

The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Doubly_Linked_Lists is called with an invalid cursor parameter. 157/2

Execution is erroneous if the list associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized. 157.1/3

Implementation Requirements

No storage associated with a doubly-linked list object shall be lost upon assignment or scope exit. 158/5

The execution of an assignment_statement for a list shall have the effect of copying the elements from the source list object to the target list object and changing the length of the target object to that of the source object. 159/3

Implementation Advice

Containers.Doubly_Linked_Lists should be implemented similarly to a linked list. In particular, if *N* is the length of a list, then the worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 should be *O*(log *N*). 160/2

The worst-case time complexity of a call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should be *O*(*N***2), and the average time complexity should be better than *O*(*N***2). 161/2

Move should not copy elements, and should minimize copying of internal data structures. 162/2

If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation. 163/2

NOTE Sorting a list never copies elements, and is a stable sort (equal elements remain in the original order). This is different than sorting an array or vector, which will often copy elements, and hence is probably not a stable sort. 164/5

A.18.4 Maps

The language-defined generic packages Containers.Hashed_Maps and Containers.Ordered_Maps provide private types Map and Cursor, and a set of operations for each type. A map container allows an arbitrary type to be used as a key to find the element associated with that key. A hashed map uses a hash function to organize the keys, while an ordered map orders the keys per a specified relation. 1/2

This subclause describes the declarations that are common to both kinds of maps. See A.18.5 for a description of the semantics specific to Containers.Hashed_Maps and A.18.6 for a description of the semantics specific to Containers.Ordered_Maps. 2/3

Static Semantics

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on map values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on map values are unspecified. 3/2

The type Map is used to represent maps. The type Map needs finalization (see 7.6). 4/2

A map contains pairs of keys and elements, called *nodes*. Map cursors designate nodes, but also can be thought of as designating an element (the element contained in the node) for consistency with the other 5/2

containers. There exists an equivalence relation on keys, whose definition is different for hashed maps and ordered maps. A map never contains two or more nodes with equivalent keys. The *length* of a map is the number of nodes it contains.

6/2 Each nonempty map has two particular nodes called the *first node* and the *last node* (which may be the same). Each node except for the last node has a *successor node*. If there are no other intervening operations, starting with the first node and repeatedly going to the successor node will visit each node in the map exactly once until the last node is reached. The exact definition of these terms is different for hashed maps and ordered maps.

7/5 Some operations check for "tampering with cursors" of a container because they depend on the set of elements of the container remaining constant, and others check for "tampering with elements" of a container because they depend on elements of the container not being replaced. When tampering with cursors is *prohibited* for a particular map object *M*, Program_Error is propagated by the finalization of *M*, as well as by a call that passes *M* to certain of the operations of this package, as indicated by the precondition of such an operation. Similarly, when tampering with elements is *prohibited* for *M*, Program_Error is propagated by a call that passes *M* to certain of the other operations of this package, as indicated by the precondition of such an operation.

Paragraphs 8 through 15 are removed as preconditions now describe these rules.

16/2 Empty_Map represents the empty Map object. It has a length of 0. If an object of type Map is not otherwise initialized, it is initialized to the same value as Empty_Map.

17/2 No_Element represents a cursor that designates no node. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

18/5 The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

19/2 Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

19.1/5 Map'Write for a Map object *M* writes Length(*M*) elements of the map to the stream. It may also write additional information about the map.

19.2/3 Map'Read reads the representation of a map from the stream, and assigns to *Item* a map with the same length and elements as was written by Map'Write.

19.3/5
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

19.4/3 Returns True if Position designates an element, and returns False otherwise.

19.5/5
```
function Has_Element (Container : Map; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

19.6/5 Returns True if Position designates an element in Container, and returns False otherwise.

20/2
```
function "=" (Left, Right : Map) return Boolean;
```

21/2 If Left and Right denote the same map object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each key *K* in Left, the function returns False if:

22/2 - a key equivalent to *K* is not present in Right; or

- the element associated with *K* in Left is not equal to the element associated with *K* in Right (using the generic formal equality operator for elements). 23/2

If the function has not returned a result after checking all of the keys, it returns True. Any exception raised during evaluation of key equivalence or element equality is propagated. 24/2

```
function Tampering_With_Cursors_Prohibited                                    24.1/5
   (Container : Map) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise. 24.2/5

```
function Tampering_With_Elements_Prohibited                                   24.3/5
   (Container : Map) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

Always returns False, regardless of whether tampering with elements is prohibited. 24.4/5

```
function Length (Container : Map) return Count_Type                           25/5
   with Nonblocking, Global => null, Use_Formal => null;
```

Returns the number of nodes in Container. 26/2

```
function Is_Empty (Container : Map) return Boolean                            27/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```

Returns True if Container is empty. 28/5

```
procedure Clear (Container : in out Map)                                      29/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

Removes all the nodes from Container. 30/2

```
function Key (Position : Cursor) return Key_Type                              31/5
   with Pre  => Position /= No_Element
                    or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Key_Type;
```

Key returns the key component of the node designated by Position. 32/5

```
function Key (Container : Map;                                                32.1/5
              Position : Cursor) return Key_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Key_Type;
```

Key returns the key component of the node designated by Position. 32.2/5

```
function Element (Position : Cursor) return Element_Type                      33/5
   with Pre  => Position /= No_Element
                    or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

Element returns the element component of the node designated by Position. 34/5

34.1/5
```
function Element (Container : Map;
                  Position  : Cursor) return Element_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

34.2/5 Element returns the element component of the node designated by Position.

35/5
```
procedure Replace_Element (Container : in out Map;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

36/5 Replace_Element assigns New_Item to the element of the node designated by Position. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object (including itself).

37/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Key     : in Key_Type;
                                         Element : in Element_Type))
   with Pre  => Position /= No_Element
                   or else raise Constraint_Error,
        Global => in all;
```

38/5 Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of the map that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

38.1/5
```
procedure Query_Element
  (Container : in Map;
   Position  : in Cursor;
   Process   : not null access procedure (Key     : in Key_Type;
                                          Element : in Element_Type))
  with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error);
```

38.2/5 Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

39/5
```
procedure Update_Element
  (Container : in out Map;
   Position  : in     Cursor;
   Process   : not null access procedure (Key     : in     Key_Type;
                                          Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

40/5 Update_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained. 41/2

```
type Constant_Reference_Type                                                    41.1/6
      (Element : not null access constant Element_Type) is
        limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
type Reference_Type                                                             41.2/6
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

The types Constant_Reference_Type and Reference_Type need finalization. 41.3/3

This paragraph was deleted. 41.4/5

```
function Constant_Reference (Container : aliased in Map;                        41.5/5
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a Map given a cursor. 41.6/3

Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized. 41.7/5

```
function Reference (Container : aliased in out Map;                             41.8/5
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a Map given a cursor. 41.9/3

Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized. 41.10/5

```
function Constant_Reference (Container : aliased in Map;                        41.11/5
                             Key       : in Key_Type)
   return Constant_Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                    or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a map given a key value. 41.12/3

41.13/3 Equivalent to Constant_Reference (Container, Find (Container, Key)).

41.14/5
```
function Reference (Container : aliased in out Map;
                    Key       : in Key_Type)
   return Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                   or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

41.15/3 This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a map given a key value.

41.16/3 Equivalent to Reference (Container, Find (Container, Key)).

41.17/5
```
procedure Assign (Target : in out Map; Source : in Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Source) = Length (Target);
```

41.18/3 If Target denotes the same object as Source, the operation has no effect. Otherwise, the key/element pairs of Source are copied to Target as for an assignment_statement assigning Source to Target.

42/5
```
procedure Move (Target : in out Map;
                Source : in out Map)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                   Length (Target) = Length (Source'Old) and then
                   Length (Source) = 0);
```

43/3 If Target denotes the same object as Source, then the operation has no effect. Otherwise, the operation is equivalent to Assign (Target, Source) followed by Clear (Source).

44/5
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length));
```

45/2 Insert checks if a node with a key equivalent to Key is already present in Container. If a match is found, Inserted is set to False and Position designates the element with the matching key. Otherwise, Insert allocates a new node, initializes it to Key and New_Item, and adds it to Container; Inserted is set to True and Position designates the newly-inserted node. Any exception raised during allocation is propagated and Container is not modified.

```
procedure Insert (Container : in out Map;                                    46/5
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length));
```

Insert inserts Key into Container as per the five-parameter Insert, with the difference that an element initialized by default (see 3.3.1) is inserted. 47/2

```
procedure Insert (Container : in out Map;                                    48/5
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1;
```

Insert inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then Constraint_Error is propagated. 49/2

```
procedure Include (Container : in out Map;                                   50/5
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length | Original_Length + 1);
```

Include inserts Key and New_Item into Container as per the five-parameter Insert, with the difference that if a node with a key equivalent to Key is already in the map, then this operation assigns Key and New_Item to the matching node. Any exception raised during assignment is propagated. 51/2

```
procedure Replace (Container : in out Map;                                   52/5
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;
```

Replace checks if a node with a key equivalent to Key is present in Container. If a match is found, Replace assigns Key and New_Item to the matching node; otherwise, Constraint_Error is propagated. 53/2

54/5
```
procedure Exclude (Container : in out Map;
                   Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length - 1 | Original_Length);
```

55/2 Exclude checks if a node with a key equivalent to Key is present in Container. If a match is found, Exclude removes the node from the map.

56/5
```
procedure Delete (Container : in out Map;
                  Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old - 1;
```

57/2 Delete checks if a node with a key equivalent to Key is present in Container. If a match is found, Delete removes the node from the map; otherwise, Constraint_Error is propagated.

58/5
```
procedure Delete (Container : in out Map;
                  Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Length (Container) = Length (Container)'Old - 1 and then
                Position = No_Element;
```

59/5 Delete removes the node designated by Position from the map.

60/5
```
function First (Container : Map) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

61/2 If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first node in Container.

62/5
```
function Next (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

63/2 Returns a cursor that designates the successor of the node designated by Position. If Position designates the last node, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

63.1/5
```
function Next (Container : Map;
               Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));
```

63.2/5 Returns a cursor designating the successor of the node designated by Position in Container.

```
procedure Next (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

64/5

Equivalent to Position := Next (Position).

65/2

```
procedure Next (Container : in     Map;
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

65.1/5

Equivalent to Position := Next (Container, Position).

65.2/5

```
function Find (Container : Map;
               Key       : Key_Type) return Cursor
      with Post => (if Find'Result = No_Element
                    then Has_Element (Container, Find'Result));
```

66/5

If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if a node with a key equivalent to Key is present in Container. If a match is found, a cursor designating the matching node is returned; otherwise, No_Element is returned.

67/2

```
function Element (Container : Map;
                  Key       : Key_Type) return Element_Type;
```

68/2

Equivalent to Element (Find (Container, Key)).

69/2

```
function Contains (Container : Map;
                   Key       : Key_Type) return Boolean;
```

70/2

Equivalent to Find (Container, Key) /= No_Element.

71/2

Paragraphs 72 and 73 were moved above.

```
procedure Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

74/5

Iterate calls Process.**all** with a cursor that designates each node in Container, starting with the first node and moving the cursor according to the successor relation. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

75/3

The nested package Stable provides a type Stable.Map that represents a *stable* map, which is one that cannot grow and shrink. Such a map can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary map.

75.1/5

The subprograms of the map package that have a parameter or result of type Map are included in the nested package Stable with the same specification, except that the following are omitted:

75.2/5

Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Assign, Move, Insert, Include, Clear, Delete, Exclude, (for Ordered_Maps) Delete_First and Delete_Last, and (for Hashed_Maps) Reserve_Capacity

75.3/5

The operations of this package are equivalent to those for ordinary maps, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True.

75.4/5

75.5/5 If a stable map is declared with the Base discriminant designating a pre-existing ordinary map, the stable map represents a stabilized view of the underlying ordinary map, and any operation on the stable map is reflected on the underlying ordinary map. While a stabilized view exists, any operation that tampers with elements performed on the underlying map is prohibited. The finalization of a stable map that provides such a view removes this restriction on the underlying ordinary map (though some other restriction can exist due to other concurrent iterations or stabilized views).

75.6/5 If a stable map is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable map, typically a call on Copy, determines the Length of the map. The Length of a stable map never changes after initialization.

Bounded (Run-Time) Errors

75.7/3 It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of a map package, to tamper with elements of any map parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the map either prior to, or subsequent to, some or all of the modifications to the map.

75.8/3 It is a bounded error to call any subprogram declared in the visible part of a map package when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

Erroneous Execution

76/2 A Cursor value is *invalid* if any of the following have occurred since it was created:

77/2 • The map that contains the node it designates has been finalized;

77.1/3 • The map that contains the node it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

78/2 • The map that contains the node it designates has been used as the Source or Target of a call to Move; or

79/3 • The node it designates has been removed from the map that previously contained the node.

80/2 The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Hashed_Maps or Containers.Ordered_Maps is called with an invalid cursor parameter.

80.1/3 Execution is erroneous if the map associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

Implementation Requirements

81/5 No storage associated with a map object shall be lost upon assignment or scope exit.

82/3 The execution of an assignment_statement for a map shall have the effect of copying the elements from the source map object to the target map object and changing the length of the target object to that of the source object.

Implementation Advice

83/2 Move should not copy elements, and should minimize copying of internal data structures.

84/2 If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation.

A.18.5 The Generic Package Containers.Hashed_Maps

Static Semantics

The generic library package Containers.Hashed_Maps has the following declaration: 1/2

```
with Ada.Iterator_Interfaces;                                                    2/5
generic
   type Key_Type is private;
   type Element_Type is private;
   with function Hash (Key : Key_Type) return Hash_Type;
   with function Equivalent_Keys (Left, Right : Key_Type)
      return Boolean;
   with function "=" (Left, Right : Element_Type)
      return Boolean is <>;
package Ada.Containers.Hashed_Maps
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is

   type Map is tagged private                                                    3/5
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Map,
           Aggregate         => (Empty     => Empty,
                                 Add_Named => Insert),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (Map) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Map)) and then
              (not Tampering_With_Elements_Prohibited (Map)),
           Preelaborable_Initialization;

   type Cursor is private                                                        4/5
      with Preelaborable_Initialization;

   Empty_Map : constant Map;                                                     5/2

   No_Element : constant Cursor;                                                 6/2

   function Has_Element (Position : Cursor) return Boolean                       6.1/5
      with Nonblocking, Global => in all, Use_Formal => null;

   function Has_Element (Container : Map; Position : Cursor)                     6.2/5
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

   package Map_Iterator_Interfaces is new                                        6.3/5
       Ada.Iterator_Interfaces (Cursor, Has_Element);

   function "=" (Left, Right : Map) return Boolean;                              7/2

   function Tampering_With_Cursors_Prohibited                                    7.1/5
      (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

   function Tampering_With_Elements_Prohibited                                   7.2/5
      (Container : Map) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

   function Empty (Capacity : Count_Type := implementation-defined)              7.3/5
      return Map
      with Post =>
              Capacity (Empty'Result) >= Capacity and then
              not Tampering_With_Elements_Prohibited (Empty'Result) and then
              not Tampering_With_Cursors_Prohibited (Empty'Result) and then
              Length (Empty'Result) = 0;
```

```
8/5         function Capacity (Container : Map) return Count_Type
               with Nonblocking, Global => null, Use_Formal => null;

9/5         procedure Reserve_Capacity (Container : in out Map;
                                        Capacity  : in     Count_Type)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                            or else raise Program_Error,
                    Post => Container.Capacity >= Capacity;

10/5        function Length (Container : Map) return Count_Type
               with Nonblocking, Global => null, Use_Formal => null;

11/5        function Is_Empty (Container : Map) return Boolean
               with Nonblocking, Global => null, Use_Formal => null,
                    Post => Is_Empty'Result = (Length (Container) = 0);

12/5        procedure Clear (Container : in out Map)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                               or else raise Program_Error,
                    Post => Capacity (Container) = Capacity (Container)'Old and then
                            Length (Container) = 0;

13/5        function Key (Position : Cursor) return Key_Type
               with Pre  => Position /= No_Element
                               or else raise Constraint_Error,
                    Nonblocking, Global => in all, Use_Formal => Key_Type;

13.1/5      function Key (Container : Map;
                          Position : Cursor) return Key_Type
               with Pre  => (Position /= No_Element
                               or else raise Constraint_Error) and then
                            (Has_Element (Container, Position)
                               or else raise Program_Error),
                    Nonblocking, Global => null, Use_Formal => Key_Type;

14/5        function Element (Position : Cursor) return Element_Type
               with Pre  => Position /= No_Element
                               or else raise Constraint_Error,
                    Nonblocking, Global => in all, Use_Formal => Element_Type;

14.1/5      function Element (Container : Map;
                              Position  : Cursor) return Element_Type
               with Pre  => (Position /= No_Element
                               or else raise Constraint_Error) and then
                            (Has_Element (Container, Position)
                              or else raise Program_Error),
                    Nonblocking, Global => null, Use_Formal => Element_Type;

15/5        procedure Replace_Element (Container : in out Map;
                                       Position  : in     Cursor;
                                       New_item  : in     Element_Type)
               with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                              or else raise Program_Error) and then
                            (Position /= No_Element
                              or else raise Constraint_Error) and then
                            (Has_Element (Container, Position)
                              or else raise Program_Error);

16/5        procedure Query_Element
              (Position : in Cursor;
               Process  : not null access procedure (Key     : in Key_Type;
                                                     Element : in Element_Type))
               with Pre  => Position /= No_Element
                               or else raise Constraint_Error,
                    Global => in all;
```

```
procedure Query_Element                                                    16.1/5
  (Container : in Map;
   Position  : in Cursor;
   Process   : not null access procedure (Key     : in Key_Type;
                                          Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);

procedure Update_Element                                                   17/5
  (Container : in out Map;
   Position  : in     Cursor;
   Process   : not null access procedure
                  (Key     : in     Key_Type;
                   Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);

type Constant_Reference_Type                                               17.1/6
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);

type Reference_Type                                                        17.2/6
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);

function Constant_Reference (Container : aliased in Map;                   17.3/5
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;

function Reference (Container : aliased in out Map;                        17.4/5
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;

function Constant_Reference (Container : aliased in Map;                   17.5/5
                             Key       : in Key_Type)
   return Constant_Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                    or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;

function Reference (Container : aliased in out Map;                        17.6/5
                    Key       : in Key_Type)
   return Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                    or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

17.7/5
```
procedure Assign (Target : in out Map; Source : in Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                  or else raise Program_Error,
        Post => Length (Source) = Length (Target) and then
                Capacity (Target) >= Length (Source);
```

17.8/5
```
function Copy (Source : Map; Capacity : Count_Type := 0)
   return Map
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                 or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

18/5
```
procedure Move (Target : in out Map;
                Source : in out Map)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                   Length (Target) = Length (Source'Old) and then
                   Length (Source) = 0);
```

19/5
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);
```

20/5
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);
```

```
procedure Insert (Container : in out Map;                                    21/5
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1 and then
                Capacity (Container) >= Length (Container);

procedure Include (Container : in out Map;                                   22/5
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length | Original_Length + 1) and then
                 Capacity (Container) >= Length (Container);

procedure Replace (Container : in out Map;                                   23/5
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;

procedure Exclude (Container : in out Map;                                   24/5
                   Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length - 1 | Original_Length);

procedure Delete (Container : in out Map;                                    25/5
                  Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old - 1;

procedure Delete (Container : in out Map;                                    26/5
                  Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Post => Length (Container) = Length (Container)'Old - 1 and then
                Position = No_Element;

function First (Container : Map) return Cursor                               27/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);

function Next (Position : Cursor) return Cursor                              28/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

28.1/5
```
      function Next (Container : Map;
                     Position : Cursor) return Cursor
         with Nonblocking, Global => null, Use_Formal => null,
              Pre  => Position = No_Element or else
                      Has_Element (Container, Position)
                         or else raise Program_Error,
              Post => (if Position = No_Element then Next'Result = No_Element
                       elsif Next'Result = No_Element then
                         Position = Last (Container)
                       else Has_Element (Container, Next'Result));
```

29/5
```
      procedure Next (Position : in out Cursor)
         with Nonblocking, Global => in all, Use_Formal => null;
```

29.1/5
```
      procedure Next (Container : in     Map;
                      Position  : in out Cursor)
         with Nonblocking, Global => null, Use_Formal => null,
              Pre  => Position = No_Element or else
                      Has_Element (Container, Position)
                         or else raise Program_Error,
              Post => (if Position /= No_Element
                       then Has_Element (Container, Position));
```

30/5
```
      function Find (Container : Map;
                     Key       : Key_Type)
         return Cursor
         with Post => (if Find'Result /= No_Element
                       then Has_Element (Container, Find'Result));
```

31/2
```
      function Element (Container : Map;
                        Key       : Key_Type)
         return Element_Type;
```

32/2
```
      function Contains (Container : Map;
                         Key       : Key_Type) return Boolean;
```

33/3 *This paragraph was deleted.*

34/5
```
      function Equivalent_Keys (Left, Right : Cursor)
         return Boolean
         with Pre    => (Left /= No_Element and then Right /= No_Element)
                           or else raise Constraint_Error,
              Global => in all;
```

35/5
```
      function Equivalent_Keys (Left  : Cursor;
                                Right : Key_Type)
         return Boolean
         with Pre    => Left /= No_Element or else raise Constraint_Error,
              Global => in all;
```

36/5
```
      function Equivalent_Keys (Left  : Key_Type;
                                Right : Cursor)
         return Boolean
         with Pre    => Right /= No_Element or else raise Constraint_Error,
              Global => in all;
```

37/5
```
      procedure Iterate
        (Container : in Map;
         Process   : not null access procedure (Position : in Cursor))
         with Allows_Exit;
```

37.1/5
```
      function Iterate (Container : in Map)
         return Map_Iterator_Interfaces.Parallel_Iterator'Class
         with Post => Tampering_With_Cursors_Prohibited (Container);
```

37.2/5
```
      package Stable is
```

```
      type Map (Base : not null access Hashed_Maps.Map) is                    37.3/5
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Variable_Indexing => Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Length),
              Global            => null,
              Default_Initial_Condition => Length (Map) = 0,
              Preelaborable_Initialization;

      type Cursor is private                                                   37.4/5
      with Preelaborable_Initialization;

      Empty_Map : constant Map;                                                37.5/5

      No_Element : constant Cursor;                                            37.6/5

      function Has_Element (Position : Cursor) return Boolean                  37.7/5
         with Nonblocking, Global => in all, Use_Formal => null;

      package Map_Iterator_Interfaces is new                                   37.8/5
         Ada.Iterator_Interfaces (Cursor, Has_Element);

      procedure Assign (Target : in out Hashed_Maps.Map;                       37.9/5
                        Source : in Map)
         with Post => Length (Source) = Length (Target);

      function Copy (Source : Hashed_Maps.Map) return Map                      37.10/5
         with Post => Length (Copy'Result) = Length (Source);

      type Constant_Reference_Type                                             37.11/6
            (Element : not null access constant Element_Type) is
              limited private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);

      type Reference_Type                                                      37.12/6
            (Element : not null access Element_Type) is limited private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);

      -- Additional subprograms as described in the text                      37.13/5
      -- are declared here.

   private                                                                     37.14/5

      ... -- not specified by the language                                     37.15/5

   end Stable;                                                                 37.16/5

private                                                                        38/2

   ... -- not specified by the language                                        39/2

end Ada.Containers.Hashed_Maps;                                                40/2
```

An object of type Map contains an expandable hash table, which is used to provide direct access to nodes. The *capacity* of an object of type Map is the maximum number of nodes that can be inserted into the hash table prior to it being automatically expanded. 41/2

Two keys *K1* and *K2* are defined to be *equivalent* if Equivalent_Keys (*K1*, *K2*) returns True. 42/2

The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular key value. For any two equivalent key values, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified. 43/2

44/2 The actual function for the generic formal function Equivalent_Keys on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Keys behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Keys, and how many times they call it, is unspecified.

45/2 If the value of a key stored in a node of a map is changed other than by an operation in this package such that at least one of Hash or Equivalent_Keys give different results, the behavior of this package is unspecified.

46/2 Which nodes are the first node and the last node of a map, and which node is the successor of a given node, are unspecified, other than the general semantics described in A.18.4.

46.1/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Map
   with Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Elements_Prohibited (Empty'Result) and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;
```

46.2/2 Returns an empty map.

47/5
```
function Capacity (Container : Map) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

48/2 Returns the capacity of Container.

49/5
```
procedure Reserve_Capacity (Container : in out Map;
                            Capacity  : in     Count_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                 or else raise Program_Error,
        Post => Container.Capacity >= Capacity;
```

50/2 Reserve_Capacity allocates a new hash table such that the length of the resulting map can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the nodes in Container onto the new hash table. It replaces the old hash table with the new hash table, and then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified.

51/5 *This paragraph was deleted.*

52/5
```
procedure Clear (Container : in out Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Capacity (Container) = Capacity (Container)'Old and then
                Length (Container) = 0;
```

53/2 In addition to the semantics described in A.18.4, Clear does not affect the capacity of Container.

53.1/5
```
procedure Assign (Target : in out Map; Source : in Map)
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error,
           Post => Length (Source) = Length (Target) and then
                   Capacity (Target) >= Length (Source);
```

53.2/3 In addition to the semantics described in A.18.4, if the length of Source is greater than the capacity of Target, Reserve_Capacity (Target, Length (Source)) is called before assigning any elements.

```
function Copy (Source : Map; Capacity : Count_Type := 0)                    53.3/5
   return Map
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

Returns a map whose keys and elements are initialized from the keys and elements of Source. 53.4/5

```
procedure Insert (Container : in out Map;                                   54/5
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                   (if Inserted then
                      Length (Container) = Original_Length + 1
                    else
                      Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);
```

In addition to the semantics described in A.18.4, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value. 55/2

```
function Equivalent_Keys (Left, Right : Cursor)                             56/5
   return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

Equivalent to Equivalent_Keys (Key (Left), Key (Right)). 57/2

```
function Equivalent_Keys (Left  : Cursor;                                   58/5
                          Right : Key_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```

Equivalent to Equivalent_Keys (Key (Left), Right). 59/2

```
function Equivalent_Keys (Left  : Key_Type;                                 60/5
                          Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
        Global => in all;
```

Equivalent to Equivalent_Keys (Left, Key (Right)). 61/2

```
function Iterate (Container : in Map)                                       61.1/5
   return Map_Iterator_Interfaces.Parallel_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor according to the successor relation when used as a forward iterator, and processing all nodes 61.2/5

concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

Implementation Advice

62/2 If *N* is the length of a map, the average time complexity of the subprograms Element, Insert, Include, Replace, Delete, Exclude, and Find that take a key parameter should be *O*(log *N*). The average time complexity of the subprograms that take a cursor parameter should be *O*(1). The average time complexity of Reserve_Capacity should be *O*(*N*).

A.18.6 The Generic Package Containers.Ordered_Maps

Static Semantics

1/2 The generic library package Containers.Ordered_Maps has the following declaration:

2/5
```
with Ada.Iterator_Interfaces;
generic
   type Key_Type is private;
   type Element_Type is private;
   with function "<" (Left, Right : Key_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Maps
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

3/5
```
   function Equivalent_Keys (Left, Right : Key_Type) return Boolean
      is (not ((Left < Right) or (Right < Left)));
```

4/5
```
   type Map is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Map,
           Aggregate         => (Empty     => Empty,
                                 Add_Named => Insert),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Length (Map) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Map)) and then
              (not Tampering_With_Elements_Prohibited (Map)),
           Preelaborable_Initialization;
```

5/5
```
   type Cursor is private
      with Preelaborable_Initialization;
```

6/2
```
   Empty_Map : constant Map;
```

7/2
```
   No_Element : constant Cursor;
```

7.1/5
```
   function Has_Element (Position : Cursor) return Boolean
      with Nonblocking, Global => in all, Use_Formal => null;
```

7.2/5
```
   function Has_Element (Container : Map; Position : Cursor)
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

7.3/5
```
   package Map_Iterator_Interfaces is new
       Ada.Iterator_Interfaces (Cursor, Has_Element);
```

8/2
```
   function "=" (Left, Right : Map) return Boolean;
```

```
function Tampering_With_Cursors_Prohibited                                    8.1/5
   (Container : Map) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;

function Tampering_With_Elements_Prohibited                                   8.2/5
   (Container : Map) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;

function Empty return Map                                                     8.3/5
   is (Empty_Map)
   with Post =>
           not Tampering_With_Elements_Prohibited (Empty'Result) and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;

function Length (Container : Map) return Count_Type                           9/5
   with Nonblocking, Global => null, Use_Formal => null;

function Is_Empty (Container : Map) return Boolean                            10/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);

procedure Clear (Container : in out Map)                                      11/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;

function Key (Position : Cursor) return Key_Type                              12/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Key_Type;

function Key (Container : Map;                                                12.1/5
              Position : Cursor) return Key_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Key_Type;

function Element (Position : Cursor) return Element_Type                      13/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => null, Use_Formal => Element_Type;

function Element (Container : Map;                                            13.1/5
                  Position  : Cursor) return Element_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;

procedure Replace_Element (Container : in out Map;                            14/5
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);

procedure Query_Element                                                       15/5
  (Position : in Cursor;
   Process  : not null access procedure (Key     : in Key_Type;
                                         Element : in Element_Type))
   with Pre  => Position /= No_Element
                    or else raise Constraint_Error,
        Global => in all;
```

15.1/5
```
   procedure Query_Element
     (Container : in Map;
      Position  : in Cursor;
      Process   : not null access procedure (Key     : in Key_Type;
                                             Element : in Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error);
```

16/5
```
   procedure Update_Element
     (Container : in out Map;
      Position  : in     Cursor;
      Process   : not null access procedure
                       (Key     : in     Key_Type;
                        Element : in out Element_Type))
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                    (Has_Element (Container, Position)
                       or else raise Program_Error);
```

16.1/6
```
   type Constant_Reference_Type
         (Element : not null access constant Element_Type) is
            limited private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);
```

16.2/6
```
   type Reference_Type
         (Element : not null access Element_Type) is limited private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);
```

16.3/5
```
   function Constant_Reference (Container : aliased in Map;
                                Position  : in Cursor)
      return Constant_Reference_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;
```

16.4/5
```
   function Reference (Container : aliased in out Map;
                       Position  : in Cursor)
      return Reference_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                      or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;
```

16.5/5
```
   function Constant_Reference (Container : aliased in Map;
                                Key       : in Key_Type)
      return Constant_Reference_Type
      with Pre  => Find (Container, Key) /= No_Element
                       or else raise Constraint_Error,
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;
```

16.6/5
```
   function Reference (Container : aliased in out Map;
                       Key       : in Key_Type)
      return Reference_Type
      with Pre  => Find (Container, Key) /= No_Element
                       or else raise Constraint_Error,
           Post => Tampering_With_Cursors_Prohibited (Container),
           Nonblocking, Global => null, Use_Formal => null;
```

```
procedure Assign (Target : in out Map; Source : in Map)                    16.7/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target);

function Copy (Source : Map)                                               16.8/5
   return Map
   with Post =>
      Length (Copy'Result) = Length (Source) and then
      not Tampering_With_Elements_Prohibited (Copy'Result) and then
      not Tampering_With_Cursors_Prohibited (Copy'Result);

procedure Move (Target : in out Map;                                       17/5
                Source : in out Map)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Length (Target) = Length (Source'Old) and then
                    Length (Source) = 0);

procedure Insert (Container : in out Map;                                  18/5
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                     Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length));

procedure Insert (Container : in out Map;                                  19/5
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                     Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length));

procedure Insert (Container : in out Map;                                  20/5
                  Key       : in     Key_Type;
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                    or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1;
```

```
21/5        procedure Include (Container : in out Map;
                               Key       : in     Key_Type;
                               New_Item  : in     Element_Type)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error) and then
                            (Length (Container) <= Count_Type'Last - 1
                                or else raise Constraint_Error),
                    Post => (declare
                               Original_Length : constant Count_Type :=
                                  Length (Container)'Old;
                             begin
                               Length (Container)
                                  in Original_Length | Original_Length + 1);

22/5        procedure Replace (Container : in out Map;
                               Key       : in     Key_Type;
                               New_Item  : in     Element_Type)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error,
                    Post => Length (Container) = Length (Container)'Old;

23/5        procedure Exclude (Container : in out Map;
                               Key       : in     Key_Type)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error,
                    Post => (declare
                               Original_Length : constant Count_Type :=
                                  Length (Container)'Old;
                             begin
                               Length (Container)
                                  in Original_Length - 1 | Original_Length);

24/5        procedure Delete (Container : in out Map;
                              Key       : in     Key_Type)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error,
                    Post => Length (Container) = Length (Container)'Old - 1;

25/5        procedure Delete (Container : in out Map;
                              Position  : in out Cursor)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error) and then
                            (Position /= No_Element
                                or else raise Constraint_Error) and then
                            (Has_Element (Container, Position)
                               or else raise Program_Error),
                    Post => Length (Container) = Length (Container)'Old - 1 and then
                            Position = No_Element;

26/5        procedure Delete_First (Container : in out Map)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error,
                    Post => (declare
                               Original_Length : constant Count_Type :=
                                  Length (Container)'Old;
                             begin
                               (if Original_Length = 0 then Length (Container) = 0
                                else Length (Container) = Original_Length - 1));

27/5        procedure Delete_Last (Container : in out Map)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error,
                    Post => (declare
                               Original_Length : constant Count_Type :=
                                  Length (Container)'Old;
                             begin
                               (if Original_Length = 0 then Length (Container) = 0
                                else Length (Container) = Original_Length - 1));
```

```
function First (Container : Map) return Cursor                                    28/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);

function First_Element (Container : Map) return Element_Type                      29/5
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);

function First_Key (Container : Map) return Key_Type                              30/5
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);

function Last (Container : Map) return Cursor                                     31/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);

function Last_Element (Container : Map) return Element_Type                       32/5
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);

function Last_Key (Container : Map) return Key_Type                               33/5
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);

function Next (Position : Cursor) return Cursor                                   34/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);

function Next (Container : Map;                                                   34.1/5
               Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));

procedure Next (Position : in out Cursor)                                         35/5
   with Nonblocking, Global => in all, Use_Formal => null;

procedure Next (Container : in     Map;                                           35.1/5
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));

function Previous (Position : Cursor) return Cursor                               36/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                    Previous'Result = No_Element);

function Previous (Container : Map;                                               36.1/5
                   Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else Has_Element (Container, Previous'Result));
```

37/5
```
      procedure Previous (Position : in out Cursor)
         with Nonblocking, Global => in all, Use_Formal => null;
```

37.1/5
```
      procedure Previous (Container : in     Map;
                          Position  : in out Cursor)
         with Nonblocking, Global => null, Use_Formal => null,
              Pre  => Position = No_Element or else
                      Has_Element (Container, Position)
                         or else raise Program_Error,
              Post => (if Position /= No_Element
                       then Has_Element (Container, Position));
```

38/5
```
      function Find (Container : Map;
                     Key       : Key_Type) return Cursor
         with Post => (if Find'Result /= No_Element
                       then Has_Element (Container, Find'Result));
```

39/2
```
      function Element (Container : Map;
                        Key       : Key_Type) return Element_Type;
```

40/5
```
      function Floor (Container : Map;
                      Key       : Key_Type) return Cursor
         with Post => (if Floor'Result /= No_Element
                       then Has_Element (Container, Floor'Result));
```

41/5
```
      function Ceiling (Container : Map;
                        Key       : Key_Type) return Cursor
         with Post => (if Ceiling'Result /= No_Element
                       then Has_Element (Container, Ceiling'Result));
```

42/2
```
      function Contains (Container : Map;
                         Key       : Key_Type) return Boolean;
```

43/3 *This paragraph was deleted.*

44/5
```
         function "<" (Left, Right : Cursor) return Boolean
            with Pre    => (Left /= No_Element and then Right /= No_Element)
                              or else raise Constraint_Error,
                 Global => in all;
```

45/5
```
         function ">" (Left, Right : Cursor) return Boolean
            with Pre    => (Left /= No_Element and then Right /= No_Element)
                              or else raise Constraint_Error,
                 Global => in all;
```

46/5
```
         function "<" (Left : Cursor; Right : Key_Type) return Boolean
            with Pre    => Left /= No_Element or else raise Constraint_Error,
                 Global => in all;
```

47/5
```
         function ">" (Left : Cursor; Right : Key_Type) return Boolean
            with Pre    => Left /= No_Element or else raise Constraint_Error,
                 Global => in all;
```

48/5
```
         function "<" (Left : Key_Type; Right : Cursor) return Boolean
            with Pre    => Right /= No_Element or else raise Constraint_Error,
                 Global => in all;
```

49/5
```
         function ">" (Left : Key_Type; Right : Cursor) return Boolean
            with Pre    => Right /= No_Element or else raise Constraint_Error,
                 Global => in all;
```

50/5
```
         procedure  Iterate
           (Container : in Map;
            Process   : not null access procedure (Position : in Cursor))
            with Allows_Exit;
```

51/5
```
         procedure Reverse_Iterate
           (Container : in Map;
            Process   : not null access procedure (Position : in Cursor))
            with Allows_Exit;
```

51.1/5
```
         function Iterate (Container : in Map)
            return Map_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
            with Post => Tampering_With_Cursors_Prohibited (Container);
```

```
      function Iterate (Container : in Map; Start : in Cursor)                    51.2/5
         return Map_Iterator_Interfaces.Reversible_Iterator'Class
         with Pre  => (Start /= No_Element
                          or else raise Constraint_Error) and then
                       (Has_Element (Container, Start)
                          or else raise Program_Error),
              Post => Tampering_With_Cursors_Prohibited (Container);

      package Stable is                                                           51.3/5

         type Map (Base : not null access Ordered_Maps.Map) is                    51.4/5
            tagged limited private
            with Constant_Indexing => Constant_Reference,
                 Variable_Indexing => Reference,
                 Default_Iterator  => Iterate,
                 Iterator_Element  => Element_Type,
                 Stable_Properties => (Length),
                 Global            => null,
                 Default_Initial_Condition => Length (Map) = 0,
                 Preelaborable_Initialization;

         type Cursor is private                                                   51.5/5
            with Preelaborable_Initialization;

         Empty_Map : constant Map;                                                51.6/5

         No_Element : constant Cursor;                                            51.7/5

         function Has_Element (Position : Cursor) return Boolean                  51.8/5
            with Nonblocking, Global => in all, Use_Formal => null;

         package Map_Iterator_Interfaces is new                                   51.9/5
            Ada.Iterator_Interfaces (Cursor, Has_Element);

         procedure Assign (Target : in out Ordered_Maps.Map;                      51.10/5
                           Source : in Map)
            with Post => Length (Source) = Length (Target);

         function Copy (Source : Ordered_Maps.Map) return Map                     51.11/5
            with Post => Length (Copy'Result) = Length (Source);

         type Constant_Reference_Type                                             51.12/6
               (Element : not null access constant Element_Type) is
                  limited private
            with Implicit_Dereference => Element,
                 Nonblocking, Global => null, Use_Formal => null,
                 Default_Initial_Condition => (raise Program_Error);

         type Reference_Type                                                      51.13/6
               (Element : not null access Element_Type) is limited private
            with Implicit_Dereference => Element,
                 Nonblocking, Global => null, Use_Formal => null,
                 Default_Initial_Condition => (raise Program_Error);

         -- Additional subprograms as described in the text                      51.14/5
         -- are declared here.

      private                                                                     51.15/5

         ... -- not specified by the language                                     51.16/5

      end Stable;                                                                 51.17/5

   private                                                                        52/2

      ... -- not specified by the language                                        53/2

   end Ada.Containers.Ordered_Maps;                                               54/2
```

Two keys *K1* and *K2* are *equivalent* if both *K1* < *K2* and *K2* < *K1* return False, using the generic formal "<" operator for keys. Function Equivalent_Keys returns True if Left and Right are equivalent, and False otherwise. 55/2

The actual function for the generic formal function "<" on Key_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict weak ordering 56/3

relationship (see A.18). If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

57/2 If the value of a key stored in a map is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.

58/3 The *first node* of a nonempty map is the one whose key is less than the key of all the other nodes in the map. The *last node* of a nonempty map is the one whose key is greater than the key of all the other elements in the map. The *successor* of a node is the node with the smallest key that is larger than the key of the given node. The *predecessor* of a node is the node with the largest key that is smaller than the key of the given node. All comparisons are done using the generic formal "<" operator for keys.

58.1/5
```
function Copy (Source : Map)
   return Map
   with Post =>
      Length (Copy'Result) = Length (Source) and then
      not Tampering_With_Elements_Prohibited (Copy'Result) and then
      not Tampering_With_Cursors_Prohibited (Copy'Result);
```

58.2/3 Returns a map whose keys and elements are initialized from the corresponding keys and elements of Source.

59/5
```
procedure Delete_First (Container : in out Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                    (if Original_Length = 0 then Length (Container) = 0
                     else Length (Container) = Original_Length - 1));
```

60/3 If Container is empty, Delete_First has no effect. Otherwise, the node designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container.

61/5
```
procedure Delete_Last (Container : in out Map)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                    Original_Length : constant Count_Type :=
                       Length (Container)'Old;
                 begin
                    (if Original_Length = 0 then Length (Container) = 0
                     else Length (Container) = Original_Length - 1));
```

62/3 If Container is empty, Delete_Last has no effect. Otherwise, the node designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

63/5
```
function First_Element (Container : Map) return Element_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);
```

64/2 Equivalent to Element (First (Container)).

65/5
```
function First_Key (Container : Map) return Key_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);
```

66/2 Equivalent to Key (First (Container)).

```
function Last (Container : Map) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```
67/5

Returns a cursor that designates the last node in Container. If Container is empty, returns No_Element.
68/2

```
function Last_Element (Container : Map) return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```
69/5

Equivalent to Element (Last (Container)).
70/2

```
function Last_Key (Container : Map) return Key_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```
71/5

Equivalent to Key (Last (Container)).
72/2

```
function Previous (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                    Previous'Result = No_Element);
```
73/5

If Position equals No_Element, then Previous returns No_Element. Otherwise, Previous returns a cursor designating the predecessor node of the one designated by Position. If Position designates the first element, then Previous returns No_Element.
74/3

```
function Previous (Container : Map;
                   Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then
                   Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                   Position = First (Container)
                 else Has_Element (Container, Previous'Result));
```
74.1/5

Returns a cursor designating the predecessor of the node designated by Position in Container, if any.
74.2/5

```
procedure Previous (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```
75/5

Equivalent to Position := Previous (Position).
76/2

```
procedure Previous (Container : in     Map;
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```
76.1/5

Equivalent to Position := Previous (Container, Position).
76.2/5

77/5
```
function Floor (Container : Map;
                Key       : Key_Type) return Cursor
   with Post => (if Floor'Result /= No_Element
                 then Has_Element (Container, Floor'Result));
```

78/3 Floor searches for the last node whose key is not greater than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise, No_Element is returned.

79/5
```
function Ceiling (Container : Map;
                  Key       : Key_Type) return Cursor
   with Post => (if Ceiling'Result /= No_Element
                 then Has_Element (Container, Ceiling'Result));
```

80/3 Ceiling searches for the first node whose key is not less than Key, using the generic formal "<" operator for keys. If such a node is found, a cursor that designates it is returned. Otherwise, No_Element is returned.

81/5
```
function "<" (Left, Right : Cursor) return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

82/2 Equivalent to Key (Left) < Key (Right).

83/5
```
function ">" (Left, Right : Cursor) return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

84/2 Equivalent to Key (Right) < Key (Left).

85/5
```
function "<" (Left : Cursor; Right : Key_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```

86/2 Equivalent to Key (Left) < Right.

87/5
```
function ">" (Left : Cursor; Right : Key_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;
```

88/2 Equivalent to Right < Key (Left).

89/5
```
function "<" (Left : Key_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;
```

90/2 Equivalent to Left < Key (Right).

91/5
```
function ">" (Left : Key_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;
```

92/2 Equivalent to Key (Right) < Left.

93/5
```
procedure Reverse_Iterate
  (Container : in Map;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

94/3 Iterates over the nodes in Container as per procedure Iterate, with the difference that the nodes are traversed in predecessor order, starting with the last node.

```
function Iterate (Container : in Map)                                                    94.1/5
   return Map_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the first node and moving the cursor according to the successor relation when used as a forward iterator, and starting with the last node and moving the cursor according to the predecessor relation when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 94.2/5

```
function Iterate (Container : in Map; Start : in Cursor)                                 94.3/5
   return Map_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre  => (Start /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Start)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each node in Container, starting with the node designated by Start and moving the cursor according to the successor relation when used as a forward iterator, or moving the cursor according to the predecessor relation when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 94.4/5

Implementation Advice

If *N* is the length of a map, then the worst-case time complexity of the Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1). 95/2

A.18.7 Sets

The language-defined generic packages Containers.Hashed_Sets and Containers.Ordered_Sets provide private types Set and Cursor, and a set of operations for each type. A set container allows elements of an arbitrary type to be stored without duplication. A hashed set uses a hash function to organize elements, while an ordered set orders its element per a specified relation. 1/2

This subclause describes the declarations that are common to both kinds of sets. See A.18.8 for a description of the semantics specific to Containers.Hashed_Sets and A.18.9 for a description of the semantics specific to Containers.Ordered_Sets. 2/3

Static Semantics

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on set values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on set values are unspecified. 3/2

The type Set is used to represent sets. The type Set needs finalization (see 7.6). 4/2

5/2 A set contains elements. Set cursors designate elements. There exists an equivalence relation on elements, whose definition is different for hashed sets and ordered sets. A set never contains two or more equivalent elements. The *length* of a set is the number of elements it contains.

6/2 Each nonempty set has two particular elements called the *first element* and the *last element* (which may be the same). Each element except for the last element has a *successor element*. If there are no other intervening operations, starting with the first element and repeatedly going to the successor element will visit each element in the set exactly once until the last element is reached. The exact definition of these terms is different for hashed sets and ordered sets.

7/5 Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant and on elements of the container not being replaced. When tampering with cursors is *prohibited* for a particular set object *S*, Program_Error is propagated by the finalization of *S*, as well as by a call that passes *S* to certain of the operations of this package, as indicated by the precondition of such an operation.

Paragraphs 8 through 14 are removed as preconditions now describe these rules.

15/2 Empty_Set represents the empty Set object. It has a length of 0. If an object of type Set is not otherwise initialized, it is initialized to the same value as Empty_Set.

16/2 No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

17/5 The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

18/2 Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

18.1/5 Set'Write for a Set object *S* writes Length(*S*) elements of the set to the stream. It may also write additional information about the set.

18.2/3 Set'Read reads the representation of a set from the stream, and assigns to *Item* a set with the same length and elements as was written by Set'Write.

18.3/5
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

18.4/3 Returns True if Position designates an element, and returns False otherwise.

18.5/5
```
function Has_Element (Container : Set; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

18.6/5 Returns True if Position designates an element in Container, and returns False otherwise.

19/2
```
function "=" (Left, Right : Set) return Boolean;
```

20/2 If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equal to *E* (using the generic formal equality operator) is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equality is propagated.

```
function Equivalent_Sets (Left, Right : Set) return Boolean;
```
21/2

If Left and Right denote the same set object, then the function returns True. If Left and Right have different lengths, then the function returns False. Otherwise, for each element *E* in Left, the function returns False if an element equivalent to *E* is not present in Right. If the function has not returned a result after checking all of the elements, it returns True. Any exception raised during evaluation of element equivalence is propagated. 22/2

```
function Tampering_With_Cursors_Prohibited
   (Container : Set) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
22.1/5

Returns True if tampering with cursors is currently prohibited for Container, and returns False otherwise. 22.2/5

```
function To_Set (New_Item : Element_Type) return Set
   with Post => Length (To_Set'Result) = 1 and then
                not Tampering_with_Cursors_Prohibited (To_Set'Result);
```
23/5

Returns a set containing the single element New_Item. 24/2

```
function Length (Container : Set) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```
25/5

Returns the number of elements in Container. 26/2

```
function Is_Empty (Container : Set) return Boolean
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);
```
27/5

Returns True if Container is empty. 28/5

```
procedure Clear (Container : in out Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```
29/5

Removes all the elements from Container. 30/2

```
function Element (Position : Cursor) return Element_Type
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```
31/5

Element returns the element designated by Position. 32/5

```
function Element (Container : Set;
                  Position  : Cursor) return Element_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```
32.1/5

Element returns the element designated by Position. 32.2/5

33/5
```
procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

34/5 Replace_Element assigns New_Item to the element designated by Position. Any exception raised by the assignment is propagated. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object (including itself).

35/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element
                    or else raise Constraint_Error,
        Global => in all;
```

36/5 Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the set that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

36.1/5
```
procedure Query_Element
  (Container : in Set;
   Position  : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

36.2/5 Query_Element calls Process.**all** with the key and element from the node designated by Position as the arguments. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

36.3/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

36.4/3 The type Constant_Reference_Type needs finalization.

36.5/5 *This paragraph was deleted.*

36.6/5
```
function Constant_Reference (Container : aliased in Set;
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

36.7/3 This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a set given a cursor.

Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the cursors of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized. 36.8/5

```
procedure Assign (Target : in out Set; Source : in Set)                    36.9/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Source) = Length (Target);
```

If Target denotes the same object as Source, the operation has no effect. Otherwise, the elements of Source are copied to Target as for an assignment_statement assigning Source to Target. 36.10/3

```
procedure Move (Target : in out Set;                                       37/5
                Source : in out Set)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                   Length (Target) = Length (Source'Old) and then
                   Length (Source) = 0);
```

If Target denotes the same object as Source, then the operation has no effect. Otherwise, the operation is equivalent to Assign (Target, Source) followed by Clear (Source). 38/3

```
procedure Insert (Container : in out Set;                                  39/5
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length));
```

Insert checks if an element equivalent to New_Item is already present in Container. If a match is found, Inserted is set to False and Position designates the matching element. Otherwise, Insert adds New_Item to Container; Inserted is set to True and Position designates the newly-inserted element. Any exception raised during allocation is propagated and Container is not modified. 40/2

```
procedure Insert (Container : in out Set;                                  41/5
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1;
```

Insert inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then Constraint_Error is propagated. 42/2

43/5
```
procedure Include (Container : in out Set;
                   New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                     or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length | Original_Length + 1);
```

44/2 Include inserts New_Item into Container as per the four-parameter Insert, with the difference that if an element equivalent to New_Item is already in the set, then it is replaced. Any exception raised during assignment is propagated.

45/5
```
procedure Replace (Container : in out Set;
                   New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;
```

46/2 Replace checks if an element equivalent to New_Item is already in the set. If a match is found, that element is replaced with New_Item; otherwise, Constraint_Error is propagated.

47/5
```
procedure Exclude (Container : in out Set;
                   Item      : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container) in
                      Original_Length - 1 | Original_Length);
```

48/2 Exclude checks if an element equivalent to Item is present in Container. If a match is found, Exclude removes the element from the set.

49/5
```
procedure Delete (Container : in out Set;
                  Item      : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old - 1;
```

50/2 Delete checks if an element equivalent to Item is present in Container. If a match is found, Delete removes the element from the set; otherwise, Constraint_Error is propagated.

51/5
```
procedure Delete (Container : in out Set;
                  Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error) and then
                (Position /= No_Element
                     or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Post => Length (Container) = Length (Container)'Old - 1 and then
                Position = No_Element;
```

52/5 Delete removes the element designated by Position from the set.

```
procedure Union (Target : in out Set;                                                    53/5
                 Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

Union inserts into Target the elements of Source that are not equivalent to some element already in Target. 54/2

```
function Union (Left, Right : Set) return Set                                            55/5
   with Post => Length (Union'Result) <=
                   Length (Left) + Length (Right) and then
                not Tampering_With_Cursors_Prohibited (Union'Result);
```

Returns a set comprising all of the elements of Left, and the elements of Right that are not equivalent to some element of Left. 56/2

```
procedure Intersection (Target : in out Set;                                             57/5
                        Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

Intersection deletes from Target the elements of Target that are not equivalent to some element of Source. 58/3

```
function Intersection (Left, Right : Set) return Set                                     59/5
   with Post => Length (Intersection'Result) <=
                   Length (Left) + Length (Right) and then
                not Tampering_With_Cursors_Prohibited (Intersection'Result);
```

Returns a set comprising all the elements of Left that are equivalent to the some element of Right. 60/2

```
procedure Difference (Target : in out Set;                                               61/5
                      Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

If Target denotes the same object as Source, then Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source. 62/2

```
function Difference (Left, Right : Set) return Set                                       63/5
   with Post => Length (Difference'Result) <= Length (Left) +
                   Length (Right) and then
                not Tampering_With_Cursors_Prohibited (Difference'Result);
```

Returns a set comprising the elements of Left that are not equivalent to some element of Right. 64/2

```
procedure Symmetric_Difference (Target : in out Set;                                     65/5
                                Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

If Target denotes the same object as Source, then Symmetric_Difference clears Target. Otherwise, it deletes from Target the elements that are equivalent to some element of Source, and inserts into Target the elements of Source that are not equivalent to some element of Target. 66/2

67/5
```
function Symmetric_Difference (Left, Right : Set) return Set
   with Post => Length (Symmetric_Difference'Result) <=
                   Length (Left) + Length (Right) and then
                not Tampering_With_Cursors_Prohibited (
                   Symmetric_Difference'Result);
```

68/2 Returns a set comprising the elements of Left that are not equivalent to some element of Right, and the elements of Right that are not equivalent to some element of Left.

69/2
```
function Overlap (Left, Right : Set) return Boolean;
```

70/3 If an element of Left is equivalent to some element of Right, then Overlap returns True. Otherwise, it returns False.

71/2
```
function Is_Subset (Subset : Set;
                    Of_Set : Set) return Boolean;
```

72/3 If an element of Subset is not equivalent to some element of Of_Set, then Is_Subset returns False. Otherwise, it returns True.

73/5
```
function First (Container : Set) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);
```

74/2 If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first element in Container.

75/5
```
function Next (Position : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);
```

76/2 Returns a cursor that designates the successor of the element designated by Position. If Position designates the last element, then No_Element is returned. If Position equals No_Element, then No_Element is returned.

76.1/5
```
function Next (Container : Set;
               Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));
```

76.2/5 Returns a cursor designating the successor of the node designated by Position in Container.

77/5
```
procedure Next (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

78/2 Equivalent to Position := Next (Position).

78.1/5
```
procedure Next (Container : in     Set;
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

78.2/5 Equivalent to Position := Next (Container, Position).

This paragraph was deleted. 79/3

```
function Find (Container : Set;                                                    80/5
               Item      : Element_Type) return Cursor
      with Post => (if Find'Result /= No_Element
                    then Has_Element (Container, Find'Result));
```

If Length (Container) equals 0, then Find returns No_Element. Otherwise, Find checks if an element equivalent to Item is present in Container. If a match is found, a cursor designating the matching element is returned; otherwise, No_Element is returned. 81/2

```
function Contains (Container : Set;                                                82/2
                   Item      : Element_Type) return Boolean;
```

Equivalent to Find (Container, Item) /= No_Element. 82.1/3

Paragraphs 83 and 84 were moved above.

```
procedure Iterate                                                                  85/5
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

Iterate calls Process.**all** with a cursor that designates each element in Container, starting with the first element and moving the cursor according to the successor relation. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 86/3

Both Containers.Hashed_Set and Containers.Ordered_Set declare a nested generic package Generic_Keys, which provides operations that allow set manipulation in terms of a key (typically, a portion of an element) instead of a complete element. The formal function Key of Generic_Keys extracts a key value from an element. It is expected to return the same value each time it is called with a particular element. The behavior of Generic_Keys is unspecified if Key behaves in some other manner. 87/2

A key is expected to unambiguously determine a single equivalence class for elements. The behavior of Generic_Keys is unspecified if the formal parameters of this package behave in some other manner. 88/2

```
function Key (Position : Cursor) return Key_Type                                   89/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

Equivalent to Key (Element (Position)). 90/2

```
function Key (Container : Set;                                                     90.1/5
              Position : Cursor) return Key_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

Equivalent to Key (Element (Container, Position)). 90.2/5

The subprograms in package Generic_Keys named Contains, Find, Element, Delete, and Exclude, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key parameter is used to locate an element in the set. 91/2

92/5
```
procedure Replace (Container : in out Set;
                   Key       : in     Key_Type;
                   New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;
```

93/2 Equivalent to Replace_Element (Container, Find (Container, Key), New_Item).

94/5
```
procedure Update_Element_Preserving_Key
  (Container : in out Set;
   Position  : in     Cursor;
   Process   : not null access procedure
                                 (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

95/5 Update_Element_Preserving_Key uses Key to save the key value *K* of the element designated by Position. Update_Element_Preserving_Key then calls Process.**all** with that element as the argument. Tampering with the cursors of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. After Process.**all** returns, Update_Element_Preserving_Key checks if *K* determines the same equivalence class as that for the new element; if not, the element is removed from the set and Program_Error is propagated.

96/2 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

96.1/6
```
type Reference_Type
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

96.2/3 The type Reference_Type needs finalization.

96.3/5 *This paragraph was deleted.*

96.4/5
```
function Reference_Preserving_Key (Container : aliased in out Set;
                                   Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

96.5/3 This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read and write access to an individual element of a set given a cursor.

96.6/5 Reference_Preserving_Key uses Key to save the key value *K*; then returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the cursors of Container is prohibited while the object returned by Reference_Preserving_Key exists and has not been finalized. When the object returned by Reference_Preserving_Key is finalized, a check is made if *K* determines the same equivalence class as that for the new element; if not, the element is removed from the set and Program_Error is propagated.

```
function Constant_Reference (Container : aliased in Set;                    96.7/5
                             Key       : in Key_Type)
   return Constant_Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                   or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container);
```

This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read access to an individual element of a set given a key value. 96.8/3

Equivalent to Constant_Reference (Container, Find (Container, Key)). 96.9/3

```
function Reference_Preserving_Key (Container : aliased in out Set;          96.10/5
                                   Key       : in Key_Type)
   return Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                   or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container);
```

This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read and write access to an individual element of a set given a key value. 96.11/3

Equivalent to Reference_Preserving_Key (Container, Find (Container, Key)). 96.12/3

The nested package Stable provides a type Stable.Set that represents a *stable* set, which is one that cannot grow and shrink. Such a set can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary set. 96.13/5

The subprograms of the set package that have a parameter or result of type Set are included in the nested package Stable with the same specification, except that the following are omitted: 96.14/5

Tampering_With_Cursors_Prohibited, Assign, Move, Insert, Include, Clear, Delete, Exclude, Replace, Replace_Element, procedures Union, Intersection, Difference, and Symmetric_Difference, (for Ordered_sets) Delete_First and Delete_Last, and (for Hashed_sets) Reserve_Capacity 96.15/5

The operations of this package are equivalent to those for ordinary sets, except that the calls to Tampering_With_Cursors_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 96.16/5

If a stable set is declared with the Base discriminant designating a pre-existing ordinary set, the stable set represents a stabilized view of the underlying ordinary set, and any operation on the stable set is reflected on the underlying ordinary set. While a stabilized view exists, any operation that tampers with cursors performed on the underlying set is prohibited. The finalization of a stable set that provides such a view removes this restriction on the underlying ordinary set (though some other restriction can exist due to other concurrent iterations or stabilized views). 96.17/5

If a stable set is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable set, typically a call on Copy, determines the Length of the set. The Length of a stable set never changes after initialization. 96.18/5

Bounded (Run-Time) Errors

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of a set package, to tamper with elements of any set parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the set either prior to, or subsequent to, some or all of the modifications to the set. 96.19/3

96.20/3 It is a bounded error to call any subprogram declared in the visible part of a set package when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

Erroneous Execution

97/2 A Cursor value is *invalid* if any of the following have occurred since it was created:

98/2 • The set that contains the element it designates has been finalized;

98.1/3 • The set that contains the element it designates has been used as the Target of a call to Assign, or as the target of an assignment_statement;

99/2 • The set that contains the element it designates has been used as the Source or Target of a call to Move; or

100/3 • The element it designates has been removed from the set that previously contained the element.

101/2 The result of "=" or Has_Element is unspecified if these functions are called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Hashed_Sets or Containers.Ordered_Sets is called with an invalid cursor parameter.

101.1/3 Execution is erroneous if the set associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

Implementation Requirements

102/5 No storage associated with a set object shall be lost upon assignment or scope exit.

103/3 The execution of an assignment_statement for a set shall have the effect of copying the elements from the source set object to the target set object and changing the length of the target object to that of the source object.

Implementation Advice

104/2 Move should not copy elements, and should minimize copying of internal data structures.

105/2 If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation.

A.18.8 The Generic Package Containers.Hashed_Sets

Static Semantics

1/2 The generic library package Containers.Hashed_Sets has the following declaration:

2/5
```
with Ada.Iterator_Interfaces;
generic
   type Element_Type is private;
   with function Hash (Element : Element_Type) return Hash_Type;
   with function Equivalent_Elements (Left, Right : Element_Type)
                 return Boolean;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Hashed_Sets
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

```
type Set is tagged private                                                    3/5
   with Constant_Indexing => Constant_Reference,
        Default_Iterator  => Iterate,
        Iterator_Element  => Element_Type,
        Iterator_View     => Stable.Set,
        Aggregate         => (Empty       => Empty,
                              Add_Unnamed => Include),
        Stable_Properties => (Length,
                              Tampering_With_Cursors_Prohibited),
        Default_Initial_Condition =>
           Length (Set) = 0 and then
           (not Tampering_With_Cursors_Prohibited (Set)),
        Preelaborable_Initialization;

type Cursor is private                                                        4/5
   with Preelaborable_Initialization;

Empty_Set : constant Set;                                                     5/2

No_Element : constant Cursor;                                                 6/2

function Has_Element (Position : Cursor) return Boolean                       6.1/5
   with Nonblocking, Global => in all, Use_Formal => null;

function Has_Element (Container : Set; Position : Cursor)                     6.2/5
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;

package Set_Iterator_Interfaces is new                                        6.3/3
    Ada.Iterator_Interfaces (Cursor, Has_Element);

function "=" (Left, Right : Set) return Boolean;                              7/2

function Equivalent_Sets (Left, Right : Set) return Boolean;                  8/2

function Tampering_With_Cursors_Prohibited                                    8.1/5
   (Container : Set) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;

function Empty (Capacity : Count_Type := implementation-defined)              8.2/5
   return Set
   with Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;

function To_Set (New_Item : Element_Type) return Set                          9/5
   with Post => Length (To_Set'Result) = 1 and then
                not Tampering_with_Cursors_Prohibited (To_Set'Result);

function Capacity (Container : Set) return Count_Type                         10/5
   with Nonblocking, Global => null, Use_Formal => null;

procedure Reserve_Capacity (Container : in out Set;                           11/5
                            Capacity  : in     Count_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Container.Capacity >= Capacity;

function Length (Container : Set) return Count_Type                           12/5
   with Nonblocking, Global => null, Use_Formal => null;

function Is_Empty (Container : Set) return Boolean                            13/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Length (Container) = 0);

procedure Clear (Container : in out Set)                                      14/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Capacity (Container) = Capacity (Container)'Old and then
                Length (Container) = 0;

function Element (Position : Cursor) return Element_Type                      15/5
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

15.1/5
```
function Element (Container : Set;
                  Position  : Cursor) return Element_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

16/5
```
procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

17/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

17.1/5
```
procedure Query_Element
  (Container : in Set;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error);
```

17.2/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

17.3/5
```
function Constant_Reference (Container : aliased in Set;
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

17.4/5
```
procedure Assign (Target : in out Set; Source : in Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target) and then
                Capacity (Target) >= Length (Source);
```

17.5/5
```
function Copy (Source : Set; Capacity : Count_Type := 0)
   return Set
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                    or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

```
procedure Move (Target : in out Set;                                                    18/5
                Source : in out Set)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                   Length (Target) = Length (Source'Old) and then
                   Length (Source) = 0);

procedure Insert (Container : in out Set;                                               19/5
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);

procedure Insert (Container : in out Set;                                               20/5
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1 and then
                Capacity (Container) >= Length (Container);

procedure Include (Container : in out Set;                                              21/5
                   New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length | Original_Length + 1) and then
                 Capacity (Container) >= Length (Container);

procedure Replace (Container : in out Set;                                              22/5
                   New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;

procedure Exclude (Container : in out Set;                                              23/5
                   Item      : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container) in
                      Original_Length - 1 | Original_Length);
```

24/5
```
procedure Delete (Container : in out Set;
                  Item      : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old - 1;
```

25/5
```
procedure Delete (Container : in out Set;
                  Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Length (Container) = Length (Container)'Old - 1 and then
                Position = No_Element;
```

26/5
```
procedure Union (Target : in out Set;
                 Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

27/5
```
function Union (Left, Right : Set) return Set
   with Post => Length (Union'Result) <=
                    Length (Left) + Length (Right) and then
                not Tampering_With_Cursors_Prohibited (Union'Result);
```

28/2
```
function "or" (Left, Right : Set) return Set renames Union;
```

29/5
```
procedure Intersection (Target : in out Set;
                        Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

30/5
```
function Intersection (Left, Right : Set) return Set
   with Post =>
          Length (Intersection'Result) <=
             Length (Left) + Length (Right) and then
          not Tampering_With_Cursors_Prohibited (Intersection'Result);
```

31/2
```
function "and" (Left, Right : Set) return Set renames Intersection;
```

32/5
```
procedure Difference (Target : in out Set;
                      Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

33/5
```
function Difference (Left, Right : Set) return Set
   with Post =>
          Length (Difference'Result) <=
             Length (Left) + Length (Right) and then
          not Tampering_With_Cursors_Prohibited (Difference'Result);
```

34/2
```
function "-" (Left, Right : Set) return Set renames Difference;
```

35/5
```
procedure Symmetric_Difference (Target : in out Set;
                                Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

36/5
```
function Symmetric_Difference (Left, Right : Set) return Set
   with Post =>
          Length (Symmetric_Difference'Result) <=
             Length (Left) + Length (Right) and then
          not Tampering_With_Cursors_Prohibited (
             Symmetric_Difference'Result);
```

37/2
```
function "xor" (Left, Right : Set) return Set
  renames Symmetric_Difference;
```

```
function Overlap (Left, Right : Set) return Boolean;                                    38/2

function Is_Subset (Subset : Set;                                                       39/2
                    Of_Set : Set) return Boolean;

function First (Container : Set) return Cursor                                          40/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);

function Next (Position : Cursor) return Cursor                                         41/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);

function Next (Container : Set;                                                         41.1/5
               Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                   Position = Last (Container)
                 else Has_Element (Container, Next'Result));

procedure Next (Position : in out Cursor)                                               42/5
   with Nonblocking, Global => in all, Use_Formal => null;

procedure Next (Container : in     Set;                                                 42.1/5
                Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));

function Find (Container : Set;                                                         43/5
               Item      : Element_Type)
   return Cursor
   with Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));

function Contains (Container : Set;                                                     44/2
                   Item      : Element_Type) return Boolean;

This paragraph was deleted.                                                             45/3

function Equivalent_Elements (Left, Right : Cursor)                                     46/5
   return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;

function Equivalent_Elements (Left  : Cursor;                                           47/5
                              Right : Element_Type)
   return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;

function Equivalent_Elements (Left  : Element_Type;                                     48/5
                              Right : Cursor)
   return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
        Global => in all;

procedure Iterate                                                                       49/5
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

```
49.1/5          function Iterate (Container : in Set)
                   return Set_Iterator_Interfaces.Parallel_Iterator'Class
                   with Post => Tampering_With_Cursors_Prohibited (Container);

50/5            generic
                   type Key_Type (<>) is private;
                   with function Key (Element : Element_Type) return Key_Type;
                   with function Hash (Key : Key_Type) return Hash_Type;
                   with function Equivalent_Keys (Left, Right : Key_Type)
                                                  return Boolean;
                package Generic_Keys
                with Nonblocking, Global => null is

51/5               function Key (Position : Cursor) return Key_Type
                      with Pre  => Position /= No_Element or else raise Constraint_Error,
                           Global => in all;

51.1/5             function Key (Container : Set;
                                 Position : Cursor) return Key_Type
                      with Pre  => (Position = No_Element
                                       or else raise Constraint_Error) and then
                                    (Has_Element (Container, Position)
                                       or else raise Program_Error);

52/2               function Element (Container : Set;
                                     Key       : Key_Type)
                     return Element_Type;

53/5               procedure Replace (Container : in out Set;
                                      Key       : in     Key_Type;
                                      New_Item  : in     Element_Type)
                      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                       or else raise Program_Error,
                           Post => Length (Container) = Length (Container)'Old;

54/5               procedure Exclude (Container : in out Set;
                                      Key       : in     Key_Type)
                      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                       or else raise Program_Error,
                           Post => (declare
                                      Original_Length : constant Count_Type :=
                                         Length (Container)'Old;
                                    begin
                                      Length (Container)
                                         in Original_Length - 1 | Original_Length);

55/5               procedure Delete (Container : in out Set;
                                     Key       : in     Key_Type)
                      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                                       or else raise Program_Error,
                           Post => Length (Container) = Length (Container)'Old - 1;

56/6               function Find (Container : Set;
                                  Key       : Key_Type)
                      return Cursor
                      with Post => (if Find'Result /= No_Element
                                    then Has_Element (Container, Find'Result));

57/2               function Contains (Container : Set;
                                      Key       : Key_Type)
                      return Boolean;

58/5               procedure Update_Element_Preserving_Key
                     (Container : in out Set;
                      Position  : in     Cursor;
                      Process   : not null access procedure
                                       (Element : in out Element_Type))
                      with Pre  => (Position /= No_Element or else
                                       raise Constraint_Error) and then
                                    (Has_Element (Container, Position) or else
                                       raise Program_Error);
```

```
   type Reference_Type                                                          58.1/6
         (Element : not null access Element_Type) is limited private
      with Implicit_Dereference => Element,
           Nonblocking, Global => in out synchronized,
           Default_Initial_Condition => (raise Program_Error);

   function Reference_Preserving_Key (Container : aliased in out Set;           58.2/5
                                      Position  : in Cursor)
      return Reference_Type
      with Pre  => (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                       or else raise Program_Error),
           Post => Tampering_With_Cursors_Prohibited (Container);

   function Constant_Reference (Container : aliased in Set;                     58.3/5
                                Key       : in Key_Type)
      return Constant_Reference_Type
      with Pre  => Find (Container, Key) /= No_Element
                       or else raise Constraint_Error,
           Post => Tampering_With_Cursors_Prohibited (Container);

   function Reference_Preserving_Key (Container : aliased in out Set;           58.4/5
                                      Key       : in Key_Type)
      return Reference_Type
      with Pre  => Find (Container, Key) /= No_Element
                       or else raise Constraint_Error,
           Post => Tampering_With_Cursors_Prohibited (Container);

end Generic_Keys;                                                               59/2

package Stable is                                                               59.1/5

   type Set (Base : not null access Hashed_Sets.Set) is                         59.2/5
      tagged limited private
      with Constant_Indexing => Constant_Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Stable_Properties => (Length),
           Global            => null,
           Default_Initial_Condition => Length (Set) = 0,
           Preelaborable_Initialization;

   type Cursor is private                                                       59.3/5
      with Preelaborable_Initialization;

   Empty_Set : constant Set;                                                    59.4/5

   No_Element : constant Cursor;                                                59.5/5

   function Has_Element (Position : Cursor) return Boolean                      59.6/5
      with Nonblocking, Global => in all, Use_Formal => null;

   package Set_Iterator_Interfaces is new                                       59.7/5
      Ada.Iterator_Interfaces (Cursor, Has_Element);

   procedure Assign (Target : in out Hashed_Sets.Set;                           59.8/5
                     Source : in Set)
      with Post => Length (Source) = Length (Target);

   function Copy (Source : Hashed_Sets.Set) return Set                          59.9/5
      with Post => Length (Copy'Result) = Length (Source);

   type Constant_Reference_Type                                                 59.10/6
         (Element : not null access constant Element_Type) is
            limited private
      with Implicit_Dereference => Element,
           Nonblocking, Global => null, Use_Formal => null,
           Default_Initial_Condition => (raise Program_Error);

   -- Additional subprograms as described in the text                          59.11/5
   -- are declared here.

private                                                                         59.12/5
```

```
59.13/5          ... -- not specified by the language
59.14/5       end Stable;
60/2       private
61/2          ... -- not specified by the language
62/2       end Ada.Containers.Hashed_Sets;
```

63/2 An object of type Set contains an expandable hash table, which is used to provide direct access to elements. The *capacity* of an object of type Set is the maximum number of elements that can be inserted into the hash table prior to it being automatically expanded.

64/2 Two elements *E1* and *E2* are defined to be *equivalent* if Equivalent_Elements (*E1*, *E2*) returns True.

65/2 The actual function for the generic formal function Hash is expected to return the same value each time it is called with a particular element value. For any two equivalent elements, the actual for Hash is expected to return the same value. If the actual for Hash behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Hash, and how many times they call it, is unspecified.

66/2 The actual function for the generic formal function Equivalent_Elements is expected to return the same value each time it is called with a particular pair of Element values. It should define an equivalence relationship, that is, be reflexive, symmetric, and transitive. If the actual for Equivalent_Elements behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Equivalent_Elements, and how many times they call it, is unspecified.

66.1/3 If the actual function for the generic formal function "=" returns True for any pair of nonequivalent elements, then the behavior of the container function "=" is unspecified.

67/2 If the value of an element stored in a set is changed other than by an operation in this package such that at least one of Hash or Equivalent_Elements give different results, the behavior of this package is unspecified.

68/2 Which elements are the first element and the last element of a set, and which element is the successor of a given element, are unspecified, other than the general semantics described in A.18.7.

68.1/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Set
   with Post =>
           Capacity (Empty'Result) >= Capacity and then
           not Tampering_With_Cursors_Prohibited (Empty'Result) and then
           Length (Empty'Result) = 0;
```

68.2/2 Returns an empty set.

69/5
```
function Capacity (Container : Set) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```

70/2 Returns the capacity of Container.

71/5
```
procedure Reserve_Capacity (Container : in out Set;
                            Capacity  : in     Count_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                or else raise Program_Error,
        Post => Container.Capacity >= Capacity;
```

72/2 Reserve_Capacity allocates a new hash table such that the length of the resulting set can become at least the value Capacity without requiring an additional call to Reserve_Capacity, and is large enough to hold the current length of Container. Reserve_Capacity then rehashes the elements in Container onto the new hash table. It replaces the old hash table with the new hash table, and

then deallocates the old hash table. Any exception raised during allocation is propagated and Container is not modified.

This paragraph was deleted. 73/5

```
procedure Clear (Container : in out Set)                                    74/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Capacity (Container) = Capacity (Container)'Old and then
                Length (Container) = 0;
```

In addition to the semantics described in A.18.7, Clear does not affect the capacity of Container. 75/2

```
procedure Assign (Target : in out Set; Source : in Set)                     75.1/5
      with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                      or else raise Program_Error,
           Post => Length (Source) = Length (Target) and then
                   Capacity (Target) >= Length (Source);
```

In addition to the semantics described in A.18.7, if the length of Source is greater than the capacity of Target, Reserve_Capacity (Target, Length (Source)) is called before assigning any elements. 75.2/3

```
function Copy (Source : Set; Capacity : Count_Type := 0)                    75.3/5
   return Set
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

Returns a set whose elements are initialized from the elements of Source. 75.4/5

```
procedure Insert (Container : in out Set;                                   76/5
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length)) and then
                 Capacity (Container) >= Length (Container);
```

In addition to the semantics described in A.18.7, if Length (Container) equals Capacity (Container), then Insert first calls Reserve_Capacity to increase the capacity of Container to some larger value. 77/2

```
function First (Container : Set) return Cursor;                             78/2
```

If Length (Container) = 0, then First returns No_Element. Otherwise, First returns a cursor that designates the first hashed element in Container. 79/2

80/5
```
function Equivalent_Elements (Left, Right : Cursor)
   return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

81/2 Equivalent to Equivalent_Elements (Element (Left), Element (Right)).

82/5
```
function Equivalent_Elements (Left  : Cursor;
                              Right : Element_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```

83/2 Equivalent to Equivalent_Elements (Element (Left), Right).

84/5
```
function Equivalent_Elements (Left  : Element_Type;
                              Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
        Global => in all;
```

85/2 Equivalent to Equivalent_Elements (Left, Element (Right)).

85.1/5
```
function Iterate (Container : in Set)
   return Set_Iterator_Interfaces.Parallel_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

85.2/5 Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting with the first element and moving the cursor according to the successor relation when used as a forward iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

86/2 For any element *E*, the actual function for the generic formal function Generic_Keys.Hash is expected to be such that Hash (*E*) = Generic_Keys.Hash (Key (*E*)). If the actuals for Key or Generic_Keys.Hash behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Generic_Keys.Hash, and how many times they call it, is unspecified.

87/2 For any two elements *E1* and *E2*, the boolean values Equivalent_Elements (*E1*, *E2*) and Equivalent_Keys (Key (*E1*), Key (*E2*)) are expected to be equal. If the actuals for Key or Equivalent_Keys behave in some other manner, the behavior of Generic_Keys is unspecified. Which subprograms of Generic_Keys call Equivalent_Keys, and how many times they call it, is unspecified.

Implementation Advice

88/2 If *N* is the length of a set, the average time complexity of the subprograms Insert, Include, Replace, Delete, Exclude, and Find that take an element parameter should be $O(\log N)$. The average time complexity of the subprograms that take a cursor parameter should be $O(1)$. The average time complexity of Reserve_Capacity should be $O(N)$.

A.18.9 The Generic Package Containers.Ordered_Sets

Static Semantics

The generic library package Containers.Ordered_Sets has the following declaration: 1/2

```
with Ada.Iterator_Interfaces;                                                    2/5
generic
   type Element_Type is private;
   with function "<" (Left, Right : Element_Type) return Boolean is <>;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Ordered_Sets
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is

   function Equivalent_Elements (Left, Right : Element_Type) return Boolean;    3/2

   type Set is tagged private                                                    4/5
      with Constant_Indexing => Constant_Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Set,
           Aggregate         => (Empty       => Empty,
                                 Add_Unnamed => Include),
           Stable_Properties => (Length,
                                 Tampering_With_Cursors_Prohibited),
           Default_Initial_Condition =>
              Length (Set) = 0 and then
              (not Tampering_With_Cursors_Prohibited (Set)),
           Preelaborable_Initialization;

   type Cursor is private                                                       5/5
      with Preelaborable_Initialization;

   Empty_Set : constant Set;                                                    6/2

   No_Element : constant Cursor;                                                7/2

   function Has_Element (Position : Cursor) return Boolean                      7.1/5
      with Nonblocking, Global => in all, Use_Formal => null;

   function Has_Element (Container : Set; Position : Cursor)                    7.2/5
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

   package Set_Iterator_Interfaces is new                                       7.3/3
       Ada.Iterator_Interfaces (Cursor, Has_Element);

   function "=" (Left, Right : Set) return Boolean;                             8/2

   function Equivalent_Sets (Left, Right : Set) return Boolean;                 9/2

   function Tampering_With_Cursors_Prohibited                                   9.1/5
      (Container : Set) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

   function Empty return Set                                                    9.2/5
      is (Empty_Set)
      with Post =>
              not Tampering_With_Cursors_Prohibited (Empty'Result) and then
              Length (Empty'Result) = 0;

   function To_Set (New_Item : Element_Type) return Set                         10/5
      with Post => Length (To_Set'Result) = 1 and then
                   not Tampering_with_Cursors_Prohibited (To_Set'Result);

   function Length (Container : Set) return Count_Type                          11/5
      with Nonblocking, Global => null, Use_Formal => null;

   function Is_Empty (Container : Set) return Boolean                           12/5
      with Nonblocking, Global => null, Use_Formal => null,
           Post => Is_Empty'Result = (Length (Container) = 0);
```

13/5
```
procedure Clear (Container : in out Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = 0;
```

14/5
```
function Element (Position : Cursor) return Element_Type
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

14.1/5
```
function Element (Container : Set;
                  Position  : Cursor) return Element_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

15/5
```
procedure Replace_Element (Container : in out Set;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

16/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => Position /= No_Element
                    or else raise Constraint_Error,
        Global => in all;
```

16.1/5
```
procedure Query_Element
  (Container : in Set;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

16.2/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
       limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

16.3/5
```
function Constant_Reference (Container : aliased in Set;
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

16.4/5
```
procedure Assign (Target : in out Set; Source : in Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Source) = Length (Target);
```

16.5/5
```
function Copy (Source : Set) return Set
   with Post => Length (Copy'Result) = Length (Source) and then
                not Tampering_With_Cursors_Prohibited (Copy'Result);
```

```
procedure Move (Target : in out Set;                                    17/5
                Source : in out Set)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                   or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                   or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                   Length (Target) = Length (Source'Old) and then
                   Length (Source) = 0);

procedure Insert (Container : in out Set;                               18/5
                  New_Item  : in     Element_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Has_Element (Container, Position) and then
                  (if Inserted then
                     Length (Container) = Original_Length + 1
                   else
                     Length (Container) = Original_Length));

procedure Insert (Container : in out Set;                               19/5
                  New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => Length (Container) = Length (Container)'Old + 1;

procedure Include (Container : in out Set;                              20/5
                   New_Item  : in     Element_Type)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Length (Container) <= Count_Type'Last - 1
                   or else raise Constraint_Error),
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length | Original_Length + 1);

procedure Replace (Container : in out Set;                              21/5
                   New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old;

procedure Exclude (Container : in out Set;                              22/5
                   Item      : in     Element_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container)
                      in Original_Length - 1 | Original_Length);
```

```
23/5        procedure Delete (Container : in out Set;
                              Item      : in     Element_Type)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                              or else raise Program_Error,
                    Post => Length (Container) = Length (Container)'Old - 1;

24/5        procedure Delete (Container : in out Set;
                              Position  : in out Cursor)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                              or else raise Program_Error) and then
                            (Position /= No_Element
                              or else raise Constraint_Error) and then
                            (Has_Element (Container, Position)
                              or else raise Program_Error),
                    Post => Length (Container) = Length (Container)'Old - 1 and then
                            Position = No_Element;

25/5        procedure Delete_First (Container : in out Set)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                              or else raise Program_Error,
                    Post => (declare
                               Original_Length : constant Count_Type :=
                                  Length (Container)'Old;
                             begin
                               (if Original_Length = 0 then Length (Container) = 0
                                else Length (Container) = Original_Length - 1));

26/5        procedure Delete_Last (Container : in out Set)
               with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                              or else raise Program_Error,
                    Post => (declare
                               Original_Length : constant Count_Type :=
                                  Length (Container)'Old;
                             begin
                               (if Original_Length = 0 then Length (Container) = 0
                                else Length (Container) = Original_Length - 1));

27/5        procedure Union (Target : in out Set;
                             Source : in     Set)
               with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                              or else raise Program_Error,
                    Post => Length (Target) <= Length (Target)'Old + Length (Source);

28/5        function Union (Left, Right : Set) return Set
               with Post => Length (Union'Result) <=
                              Length (Left) + Length (Right) and then
                            not Tampering_With_Cursors_Prohibited (Union'Result);

29/2        function "or" (Left, Right : Set) return Set renames Union;

30/5        procedure Intersection (Target : in out Set;
                                    Source : in     Set)
               with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                              or else raise Program_Error,
                    Post => Length (Target) <= Length (Target)'Old + Length (Source);

31/5        function Intersection (Left, Right : Set) return Set
               with Post =>
                      Length (Intersection'Result) <=
                         Length (Left) + Length (Right) and then
                      not Tampering_With_Cursors_Prohibited (Intersection'Result);

32/2        function "and" (Left, Right : Set) return Set renames Intersection;

33/5        procedure Difference (Target : in out Set;
                                  Source : in     Set)
               with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                              or else raise Program_Error,
                    Post => Length (Target) <= Length (Target)'Old + Length (Source);
```

```
function Difference (Left, Right : Set) return Set                                    34/5
   with Post =>
           Length (Difference'Result) <=
              Length (Left) + Length (Right) and then
           not Tampering_With_Cursors_Prohibited (Difference'Result);

function "-" (Left, Right : Set) return Set renames Difference;                       35/2

procedure Symmetric_Difference (Target : in out Set;                                  36/5
                                Source : in     Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Length (Target) <= Length (Target)'Old + Length (Source);

function Symmetric_Difference (Left, Right : Set) return Set                          37/5
   with Post =>
           Length (Symmetric_Difference'Result) <=
              Length (Left) + Length (Right) and then
           not Tampering_With_Cursors_Prohibited (
              Symmetric_Difference'Result);

function "xor" (Left, Right : Set) return Set renames                                 38/2
   Symmetric_Difference;

function Overlap (Left, Right : Set) return Boolean;                                  39/2

function Is_Subset (Subset : Set;                                                     40/2
                    Of_Set : Set) return Boolean;

function First (Container : Set) return Cursor                                        41/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container)
                 then Has_Element (Container, First'Result)
                 else First'Result = No_Element);

function First_Element (Container : Set)                                              42/5
   return Element_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);

function Last (Container : Set) return Cursor                                         43/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container) then
                    Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);

function Last_Element (Container : Set)                                               44/5
   return Element_Type
   with Pre => (not Is_Empty (Container)
                    or else raise Constraint_Error);

function Next (Position : Cursor) return Cursor                                       45/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then Next'Result = No_Element);

function Next (Container : Set;                                                       45.1/5
               Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then Next'Result = No_Element
                 elsif Next'Result = No_Element then
                    Position = Last (Container)
                 else Has_Element (Container, Next'Result));

procedure Next (Position : in out Cursor)                                             46/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

46.1/5
```
      procedure Next (Container : in     Set;
                      Position  : in out Cursor)
         with Nonblocking, Global => null, Use_Formal => null,
              Pre  => Position = No_Element or else
                      Has_Element (Container, Position)
                         or else raise Program_Error,
              Post => (if Position /= No_Element
                       then Has_Element (Container, Position));
```

47/5
```
      function Previous (Position : Cursor) return Cursor
         with Nonblocking, Global => in all, Use_Formal => null,
              Post => (if Position = No_Element then
                          Previous'Result = No_Element);
```

47.1/5
```
      function Previous (Container : Set;
                         Position  : Cursor) return Cursor
         with Nonblocking, Global => null, Use_Formal => null,
              Pre  => Position = No_Element or else
                      Has_Element (Container, Position)
                         or else raise Program_Error,
              Post => (if Position = No_Element then
                          Previous'Result = No_Element
                       elsif Previous'Result = No_Element then
                          Position = First (Container)
                       else Has_Element (Container, Previous'Result));
```

48/5
```
      procedure Previous (Position : in out Cursor)
         with Nonblocking, Global => in all,
              Use_Formal => null;
```

48.1/5
```
      procedure Previous (Container : in     Set;
                          Position  : in out Cursor)
         with Nonblocking, Global => null, Use_Formal => null,
              Pre  => Position = No_Element or else
                      Has_Element (Container, Position)
                         or else raise Program_Error,
              Post => (if Position /= No_Element
                       then Has_Element (Container, Position));
```

49/5
```
      function Find (Container : Set;
                     Item      : Element_Type) return Cursor
         with Post => (if Find'Result /= No_Element
                       then Has_Element (Container, Find'Result));
```

50/2
```
      function Floor (Container : Set;
                      Item      : Element_Type) return Cursor
         with Post => (if Floor'Result /= No_Element
                       then Has_Element (Container, Floor'Result));
```

51/2
```
      function Ceiling (Container : Set;
                        Item      : Element_Type) return Cursor
         with Post => (if Ceiling'Result /= No_Element
                       then Has_Element (Container, Ceiling'Result));
```

52/2
```
      function Contains (Container : Set;
                         Item      : Element_Type) return Boolean;
```

53/3 *This paragraph was deleted.*

54/5
```
      function "<" (Left, Right : Cursor) return Boolean
         with Pre    => (Left /= No_Element and then Right /= No_Element)
                           or else raise Constraint_Error,
              Global => in all;
```

55/5
```
      function ">" (Left, Right : Cursor) return Boolean
         with Pre    => (Left /= No_Element and then Right /= No_Element)
                           or else raise Constraint_Error,
              Global => in all;
```

56/5
```
      function "<" (Left : Cursor; Right : Element_Type) return Boolean
         with Pre    => Left /= No_Element or else raise Constraint_Error,
              Global => in all;
```

```
function ">" (Left : Cursor; Right : Element_Type) return Boolean                    57/5
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;

function "<" (Left : Element_Type; Right : Cursor) return Boolean                    58/5
   with Pre    => Right /= No_Element or else raise Constraint_Error,
        Global => in all;

function ">" (Left : Element_Type; Right : Cursor) return Boolean                    59/5
   with Pre    => Right /= No_Element or else raise Constraint_Error,
        Global => in all;

procedure  Iterate                                                                   60/5
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;

procedure Reverse_Iterate                                                            61/5
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;

function Iterate (Container : in Set)                                                61.1/5
   return Set_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);

function Iterate (Container : in Set; Start : in Cursor)                             61.2/5
   return Set_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre  => (Start /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Start)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);

generic                                                                              62/5
   type Key_Type (<>) is private;
   with function Key (Element : Element_Type) return Key_Type;
   with function "<" (Left, Right : Key_Type)
      return Boolean is <>;
package Generic_Keys
with Nonblocking, Global => null is

    function Equivalent_Keys (Left, Right : Key_Type)                                63/2
       return Boolean;

   function Key (Position : Cursor) return Key_Type                                  64/5
      with Pre  => Position /= No_Element or else raise Constraint_Error,
           Global => in all;

   function Key (Container : Set;                                                    64.1/5
                 Position : Cursor) return Key_Type
      with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                   (Has_Element (Container, Position)
                    or else raise Program_Error);

    function Element (Container : Set;                                               65/2
                      Key       : Key_Type)
       return Element_Type;

   procedure Replace (Container : in out Set;                                        66/5
                      Key       : in     Key_Type;
                      New_Item  : in     Element_Type)
      with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                     or else raise Program_Error,
           Post => Length (Container) = Length (Container)'Old;
```

67/5
```
procedure Exclude (Container : in out Set;
                   Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   Length (Container) in
                      Original_Length - 1 | Original_Length);
```

68/5
```
procedure Delete (Container : in out Set;
                  Key       : in     Key_Type)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error,
        Post => Length (Container) = Length (Container)'Old - 1;
```

69/5
```
function Find (Container : Set;
               Key       : Key_Type) return Cursor
   with Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));
```

70/5
```
function Floor (Container : Set;
                Key       : Key_Type) return Cursor
   with Post => (if Floor'Result /= No_Element
                 then Has_Element (Container, Floor'Result));
```

71/5
```
function Ceiling (Container : Set;
                  Key       : Key_Type) return Cursor
   with Post => (if Ceiling'Result /= No_Element
                 then Has_Element (Container, Ceiling'Result));
```

72/2
```
function Contains (Container : Set;
                   Key       : Key_Type) return Boolean;
```

73/5
```
procedure Update_Element_Preserving_Key
  (Container : in out Set;
   Position  : in     Cursor;
   Process   : not null access procedure
                  (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error);
```

73.1/6
```
type Reference_Type
     (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

73.2/5
```
function Reference_Preserving_Key (Container : aliased in out Set;
                                   Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);;
```

73.3/5
```
function Constant_Reference (Container : aliased in Set;
                             Key       : in Key_Type)
   return Constant_Reference_Type
   with Pre  => Find (Container, Key) /= No_Element
                    or else raise Constraint_Error,
        Post => Tampering_With_Cursors_Prohibited (Container);;
```

```
      function Reference_Preserving_Key (Container : aliased in out Set;
                                         Key       : in Key_Type)
         return Reference_Type
         with Pre  => Find (Container, Key) /= No_Element
                        or else raise Constraint_Error,
              Post => Tampering_With_Cursors_Prohibited (Container);;
   end Generic_Keys;
   package Stable is
      type Set (Base : not null access Ordered_Sets.Set) is
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Length),
              Global            => null,
              Default_Initial_Condition => Length (Set) = 0,
              Preelaborable_Initialization;
      type Cursor is private
         with Preelaborable_Initialization;
      Empty_Set : constant Set;
      No_Element : constant Cursor;
      function Has_Element (Position : Cursor) return Boolean
         with Nonblocking, Global => in all, Use_Formal => null;
      package Set_Iterator_Interfaces is new
         Ada.Iterator_Interfaces (Cursor, Has_Element);
      procedure Assign (Target : in out Ordered_Sets.Set;
                        Source : in Set)
         with Post => Length (Source) = Length (Target);
      function Copy (Source : Ordered_Sets.Set) return Set
         with Post => Length (Copy'Result) = Length (Source);
      type Constant_Reference_Type
            (Element : not null access constant Element_Type) is
              limited private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null, Use_Formal => null,
              Default_Initial_Condition => (raise Program_Error);
      -- Additional subprograms as described in the text
      -- are declared here.
   private
      ... -- not specified by the language
   end Stable;
private
   ... -- not specified by the language
end Ada.Containers.Ordered_Sets;
```

Two elements *E1* and *E2* are *equivalent* if both *E1* < *E2* and *E2* < *E1* return False, using the generic formal "<" operator for elements. Function Equivalent_Elements returns True if Left and Right are equivalent, and False otherwise.

The actual function for the generic formal function "<" on Element_Type values is expected to return the same value each time it is called with a particular pair of key values. It should define a strict weak ordering relationship (see A.18). If the actual for "<" behaves in some other manner, the behavior of this package is unspecified. Which subprograms of this package call "<" and how many times they call it, is unspecified.

79.1/3 If the actual function for the generic formal function "=" returns True for any pair of nonequivalent elements, then the behavior of the container function "=" is unspecified.

80/2 If the value of an element stored in a set is changed other than by an operation in this package such that at least one of "<" or "=" give different results, the behavior of this package is unspecified.

81/3 The *first element* of a nonempty set is the one which is less than all the other elements in the set. The *last element* of a nonempty set is the one which is greater than all the other elements in the set. The *successor* of an element is the smallest element that is larger than the given element. The *predecessor* of an element is the largest element that is smaller than the given element. All comparisons are done using the generic formal "<" operator for elements.

81.1/5
```
function Copy (Source : Set) return Set
   with Post => Length (Copy'Result) = Length (Source) and then
                not Tampering_With_Cursors_Prohibited (Copy'Result);
```

81.2/3 Returns a set whose elements are initialized from the corresponding elements of Source.

82/5
```
procedure Delete_First (Container : in out Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   (if Original_Length = 0 then Length (Container) = 0
                    else Length (Container) = Original_Length - 1));
```

83/3 If Container is empty, Delete_First has no effect. Otherwise, the element designated by First (Container) is removed from Container. Delete_First tampers with the cursors of Container.

84/5
```
procedure Delete_Last (Container : in out Set)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => (declare
                   Original_Length : constant Count_Type :=
                      Length (Container)'Old;
                 begin
                   (if Original_Length = 0 then Length (Container) = 0
                    else Length (Container) = Original_Length - 1));
```

85/3 If Container is empty, Delete_Last has no effect. Otherwise, the element designated by Last (Container) is removed from Container. Delete_Last tampers with the cursors of Container.

86/5
```
function First_Element (Container : Set) return Element_Type
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

87/2 Equivalent to Element (First (Container)).

88/5
```
function Last (Container : Set) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => (if not Is_Empty (Container) then
                    Has_Element (Container, Last'Result)
                 else Last'Result = No_Element);
```

89/2 Returns a cursor that designates the last element in Container. If Container is empty, returns No_Element.

```
function Last_Element (Container : Set) return Element_Type                    90/5
   with Pre => (not Is_Empty (Container)
                   or else raise Constraint_Error);
```

Equivalent to Element (Last (Container)). 91/2

```
function Previous (Position : Cursor) return Cursor                            92/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element then
                    Previous'Result = No_Element);
```

If Position equals No_Element, then Previous returns No_Element. Otherwise, Previous returns a cursor designating the predecessor element of the one designated by Position. If Position designates the first element, then Previous returns No_Element. 93/3

```
function Previous (Container : Set;                                            93.1/5
                   Position : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position = No_Element then
                    Previous'Result = No_Element
                 elsif Previous'Result = No_Element then
                    Position = First (Container)
                 else Has_Element (Container, Previous'Result));
```

Returns a cursor designating the predecessor of the node designated by Position in Container, if any. 93.2/5

```
procedure Previous (Position : in out Cursor)                                  94/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

Equivalent to Position := Previous (Position). 95/2

```
procedure Previous (Container : in     Set;                                    95.1/5
                    Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Position = No_Element or else
                Has_Element (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

Equivalent to Position := Previous (Container, Position). 95.2/5

```
function Floor (Container : Set;                                               96/5
                Item      : Element_Type) return Cursor
   with Post => (if Floor'Result /= No_Element
                 then Has_Element (Container, Floor'Result));
```

Floor searches for the last element which is not greater than Item. If such an element is found, a cursor that designates it is returned. Otherwise, No_Element is returned. 97/3

```
function Ceiling (Container : Set;                                             98/5
                  Item      : Element_Type) return Cursor
   with Post => (if Ceiling'Result /= No_Element
                 then Has_Element (Container, Ceiling'Result));
```

Ceiling searches for the first element which is not less than Item. If such an element is found, a cursor that designates it is returned. Otherwise, No_Element is returned. 99/3

100/5
```
function "<" (Left, Right : Cursor) return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

101/2 Equivalent to Element (Left) < Element (Right).

102/5
```
function ">" (Left, Right : Cursor) return Boolean
   with Pre    => (Left /= No_Element and then Right /= No_Element)
                     or else raise Constraint_Error,
        Global => in all;
```

103/2 Equivalent to Element (Right) < Element (Left).

104/5
```
function "<" (Left : Cursor; Right : Element_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
        Global => in all;
```

105/2 Equivalent to Element (Left) < Right.

106/5
```
function ">" (Left : Cursor; Right : Element_Type) return Boolean
   with Pre    => Left /= No_Element or else raise Constraint_Error,
           Global => in all;
```

107/2 Equivalent to Right < Element (Left).

108/5
```
function "<" (Left : Element_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;
```

109/2 Equivalent to Left < Element (Right).

110/5
```
function ">" (Left : Element_Type; Right : Cursor) return Boolean
   with Pre    => Right /= No_Element or else raise Constraint_Error,
           Global => in all;
```

111/2 Equivalent to Element (Right) < Left.

112/5
```
procedure Reverse_Iterate
  (Container : in Set;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

113/3 Iterates over the elements in Container as per procedure Iterate, with the difference that the elements are traversed in predecessor order, starting with the last element.

113.1/5
```
function Iterate (Container : in Set)
   return Set_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

113.2/5 Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting with the first element and moving the cursor according to the successor relation when used as a forward iterator, and starting with the last element and moving the cursor according to the predecessor relation when used as a reverse iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

```
function Iterate (Container : in Set; Start : in Cursor)                    113.3/5
   return Set_Iterator_Interfaces.Reversible_Iterator'Class
   with Pre  => (Start /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Start)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

Iterate returns a reversible iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting with the element designated by Start and moving the cursor according to the successor relation when used as a forward iterator, or moving the cursor according to the predecessor relation when used as a reverse iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 113.4/5

For any two elements *E1* and *E2*, the boolean values (*E1* < *E2*) and (Key(*E1*) < Key(*E2*)) are expected to be equal. If the actuals for Key or Generic_Keys."<" behave in some other manner, the behavior of this package is unspecified. Which subprograms of this package call Key and Generic_Keys."<", and how many times the functions are called, is unspecified. 114/2

In addition to the semantics described in A.18.7, the subprograms in package Generic_Keys named Floor and Ceiling, are equivalent to the corresponding subprograms in the parent package, with the difference that the Key subprogram parameter is compared to elements in the container using the Key and "<" generic formal functions. The function named Equivalent_Keys in package Generic_Keys returns True if both Left < Right and Right < Left return False using the generic formal "<" operator, and returns True otherwise. 115/2

Implementation Advice

If *N* is the length of a set, then the worst-case time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations that take an element parameter should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms that take a cursor parameter should be *O*(1). 116/2

A.18.10 The Generic Package Containers.Multiway_Trees

The language-defined generic package Containers.Multiway_Trees provides private types Tree and Cursor, and a set of operations for each type. A multiway tree container is well-suited to represent nested structures. 1/3

A multiway tree container object manages a tree of *nodes*, consisting of a *root node* and a set of *internal nodes*; each internal node contains an element and pointers to the parent, first child, last child, next (successor) sibling, and previous (predecessor) sibling internal nodes. A cursor designates a particular node within a tree (and by extension the element contained in that node, if any). A cursor keeps designating the same node (and element) as long as the node is part of the container, even if the node is moved within the container. 2/4

A *subtree* is a particular node (which *roots the subtree*) and all of its child nodes (including all of the children of the child nodes, recursively). The root node is always present and has neither an associated element value nor any parent node; it has pointers to its first child and its last child, if any. The root node provides a place to add nodes to an otherwise empty tree and represents the base of the tree. 3/4

4/3 A node that has no children is called a *leaf node*. The *ancestors* of a node are the node itself, its parent node, the parent of the parent node, and so on until a node with no parent is reached. Similarly, the *descendants* of a node are the node itself, its child nodes, the children of each child node, and so on.

5/3 The nodes of a subtree can be visited in several different orders. For a *depth-first order*, after visiting a node, the nodes of its child list are each visited in depth-first order, with each child node visited in natural order (first child to last child).

Static Semantics

6/3 The generic library package Containers.Multiway_Trees has the following declaration:

7/5
```
with Ada.Iterator_Interfaces;
generic
   type Element_Type is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Multiway_Trees
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is
```

8/5
```
   type Tree is tagged private
      with Constant_Indexing => Constant_Reference,
           Variable_Indexing => Reference,
           Default_Iterator  => Iterate,
           Iterator_Element  => Element_Type,
           Iterator_View     => Stable.Tree,
           Stable_Properties => (Node_Count,
                                 Tampering_With_Cursors_Prohibited,
                                 Tampering_With_Elements_Prohibited),
           Default_Initial_Condition =>
              Node_Count (Tree) = 1 and then
              (not Tampering_With_Cursors_Prohibited (Tree)) and then
              (not Tampering_With_Elements_Prohibited (Tree)),
           Preelaborable_Initialization;
```

9/5
```
   type Cursor is private
      with Preelaborable_Initialization;
```

10/3
```
   Empty_Tree : constant Tree;
```

11/3
```
   No_Element : constant Cursor;
```

11.1/5
```
   function Equal_Element (Left, Right : Element_Type)
      return Boolean renames "=";
```

12/5
```
   function Has_Element (Position : Cursor) return Boolean
      with Nonblocking, Global => in all, Use_Formal => null;
```

12.1/5
```
   function Has_Element (Container : Tree; Position : Cursor)
      return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

13/3
```
   package Tree_Iterator_Interfaces is new
      Ada.Iterator_Interfaces (Cursor, Has_Element);
```

14/3
```
   function Equal_Subtree (Left_Position : Cursor;
                           Right_Position: Cursor) return Boolean;
```

15/3
```
   function "=" (Left, Right : Tree) return Boolean;
```

15.1/5
```
   function Tampering_With_Cursors_Prohibited
      (Container : Tree) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

15.2/5
```
   function Tampering_With_Elements_Prohibited
      (Container : Tree) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;
```

```
function Empty return Tree                                                    15.3/5
   is (Empty_Tree)
   with Post =>
         not Tampering_With_Elements_Prohibited (Empty'Result) and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Node_Count (Empty'Result) = 1;

function Is_Empty (Container : Tree) return Boolean                           16/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Node_Count (Container) = 1);

function Node_Count (Container : Tree) return Count_Type                      17/5
   with Nonblocking, Global => null, Use_Formal => null;

function Subtree_Node_Count (Position : Cursor) return Count_Type             18/5
   with Nonblocking, Global => in all, Use_Formal => null;

function Subtree_Node_Count (Container : Tree; Position : Cursor)             18.1/5
   return Count_Type
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;

function Depth (Position : Cursor) return Count_Type                          19/5
   with Nonblocking, Global => in all, Use_Formal => null;

function Depth (Container : Tree; Position : Cursor)                          19.1/5
   return Count_Type
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;

function Is_Root (Position : Cursor) return Boolean                           20/5
   with Nonblocking, Global => in all, Use_Formal => null;

function Is_Root (Container : Tree; Position : Cursor)                        20.1/5
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;

function Is_Leaf (Position : Cursor) return Boolean                           21/5
   with Nonblocking, Global => in all, Use_Formal => null;

function Is_Leaf (Container : Tree; Position : Cursor)                        21.1/5
   return Boolean
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;

function Is_Ancestor_Of (Container : Tree;                                    21.2/5
                         Parent   : Cursor;
                         Position : Cursor) return Boolean
   with Pre => (Meaningful_For (Container, Position)
                  or else raise Program_Error) and then
               (Meaningful_For (Container, Parent)
                  or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => null;

function Root (Container : Tree) return Cursor                                22/5
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Root'Result /= No_Element and then
                not Has_Element (Container, Root'Result);

function Meaningful_For (Container : Tree; Position : Cursor)                 22.1/5
   return Boolean is
   (Position = No_Element or else
    Is_Root (Container, Position) or else
    Has_Element (Container, Position))
   with Nonblocking, Global => null, Use_Formal => null;

procedure Clear (Container : in out Tree)                                     23/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                  or else raise Program_Error,
        Post => Node_Count (Container) = 1;
```

24/5
```
function Element (Position : Cursor) return Element_Type
   with Pre => (Position /= No_Element or else
                   raise Constraint_Error) and then
                (Has_Element (Position) or else raise Program_Error),
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

24.1/5
```
function Element (Container : Tree;
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element or else
                   raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

25/5
```
procedure Replace_Element (Container : in out Tree;
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

26/5
```
procedure Query_Element
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Position) or else raise Program_Error),
        Global => in all;
```

26.1/5
```
procedure Query_Element
  (Container : in Tree;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

27/5
```
procedure Update_Element
  (Container : in out Tree;
   Position  : in     Cursor;
   Process   : not null access procedure
                  (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

28/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
       limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

29/6
```
type Reference_Type
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

```
function Constant_Reference (Container : aliased in Tree;                    30/5
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;

function Reference (Container : aliased in out Tree;                         31/5
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;

procedure Assign (Target : in out Tree; Source : in Tree)                    32/5
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Node_Count (Source) = Node_Count (Target);

function Copy (Source : Tree) return Tree                                    33/5
   with Post =>
          Node_Count (Copy'Result) = Node_Count (Source) and then
          not Tampering_With_Elements_Prohibited (Copy'Result) and then
          not Tampering_With_Cursors_Prohibited (Copy'Result);

procedure Move (Target : in out Tree;                                        34/5
                Source : in out Tree)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                 Node_Count (Target) = Node_Count (Source'Old) and then
                 Node_Count (Source) = 1);

procedure Delete_Leaf (Container : in out Tree;                              35/5
                       Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error) and then
                (Is_Leaf (Container, Position)
                    or else raise Constraint_Error),
        Post =>
           Node_Count (Container)'Old = Node_Count (Container)+1 and then
           Position = No_Element;

procedure Delete_Subtree (Container : in out Tree;                           36/5
                          Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Node_Count (Container)'Old = Node_Count (Container) +
                    Subtree_Node_Count (Container, Position)'Old and then
                Position = No_Element;
```

```
37/5        procedure Swap (Container : in out Tree;
                            I, J      : in     Cursor)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error) and then
                            (I /= No_Element or else Constraint_Error) and then
                            (J /= No_Element or else Constraint_Error) and then
                            (Has_Element (Container, I)
                                or else raise Program_Error) and then
                            (Has_Element (Container, J)
                                or else raise Program_Error);

38/5        function Find (Container : Tree;
                           Item      : Element_Type)
               return Cursor
               with Post => (if Find'Result /= No_Element
                             then Has_Element (Container, Find'Result));

39/5        function Find_In_Subtree (Position : Cursor;
                                      Item     : Element_Type)
               return Cursor
               with Pre  => Position /= No_Element or else raise Constraint_Error,
                    Post => (if Find_In_Subtree'Result = No_Element
                             then Has_Element (Find_In_Subtree'Result)),
                    Global => in all;

39.1/5      function Find_In_Subtree (Container : Tree;
                                      Position  : Cursor;
                                      Item      : Element_Type)
               return Cursor
               with Pre  => (Position /= No_Element
                                or else raise Constraint_Error) and then
                            (Meaningful_For (Container, Position)
                                or else raise Program_Error),
                    Post => (if Find_In_Subtree'Result /= No_Element
                             then Has_Element (Container, Find_In_Subtree'Result));

40/5        function Ancestor_Find (Position : Cursor;
                                    Item     : Element_Type)
               return Cursor
               with Pre  => Position /= No_Element or else raise Constraint_Error,
                    Post => (if Ancestor_Find'Result = No_Element
                             then Has_Element (Ancestor_Find'Result)),
                    Global => in all;

40.1/5      function Ancestor_Find (Container : Tree;
                                    Position  : Cursor;
                                    Item      : Element_Type)
               return Cursor
               with Pre  => (Position /= No_Element
                                or else raise Constraint_Error) and then
                            (Meaningful_For (Container, Position)
                                or else raise Program_Error),
                    Post => (if Ancestor_Find'Result = No_Element
                             then Has_Element (Container, Ancestor_Find'Result));

41/3        function Contains (Container : Tree;
                               Item      : Element_Type) return Boolean;

42/5        procedure Iterate
              (Container : in Tree;
               Process   : not null access procedure (Position : in Cursor))
               with Allows_Exit;

43/5        procedure Iterate_Subtree
              (Position  : in Cursor;
               Process   : not null access procedure (Position : in Cursor))
               with Allows_Exit,
                    Pre  => Position /= No_Element or else raise Constraint_Error,
                    Global => in all;
```

```
procedure Iterate_Subtree                                                    43.1/5
  (Container : in Tree;
   Position  : in Cursor;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                    or else raise Program_Error);

function Iterate (Container : in Tree)                                       44/5
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);

function Iterate_Subtree (Position : in Cursor)                              45/5
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Pre    => Position /= No_Element or else raise Constraint_Error,
        Global => in all;

function Iterate_Subtree (Container : in Tree; Position : in Cursor)        45.1/5
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);

function Child_Count (Parent : Cursor) return Count_Type                     46/5
   with Post => (if Parent = No_Element then Child_Count'Result = 0),
   with Nonblocking, Global => in all, Use_Formal => null;

function Child_Count (Container : Tree; Parent : Cursor)                    46.1/5
   return Count_Type
   with Pre  => Meaningful_For (Container, Parent)
                    or else raise Program_Error,
        Post => (if Parent = No_Element then Child_Count'Result = 0),
        Nonblocking, Global => null, Use_Formal => null;

function Child_Depth (Parent, Child : Cursor) return Count_Type              47/5
   with Pre  => (Parent = No_Element and then Child = No_Element)
                    or else raise Constraint_Error,
   with Nonblocking, Global => in all, Use_Formal => null;

function Child_Depth (Container : Tree; Parent, Child : Cursor)             47.1/5
   return Count_Type
   with Pre  => ((Parent = No_Element and then Child = No_Element)
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                    or else raise Program_Error) and then
                (Meaningful_For (Container, Child)
                    or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => null;

procedure Insert_Child (Container : in out Tree;                             48/5
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                (Parent /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                    or else raise Program_Error) and then
                (Meaningful_For (Container, Before)
                    or else raise Program_Error) and then
                (Before = No_Element or else
                 Container.Parent (Before) = Parent
                    or else raise Constraint_Error),
        Post => Node_Count (Container) =
                Node_Count (Container)'Old + Count;
```

49/5
```
      procedure Insert_Child (Container : in out Tree;
                              Parent    : in     Cursor;
                              Before    : in     Cursor;
                              New_Item  : in     Element_Type;
                              Position  :    out Cursor;
                              Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Parent /= No_Element
                         or else raise Constraint_Error) and then
                      (Meaningful_For (Container, Parent)
                         or else raise Program_Error) and then
                      (Meaningful_For (Container, Before)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Container.Parent (Before) = Parent
                         or else raise Constraint_Error),
              Post => (Node_Count (Container) =
                       Node_Count (Container)'Old + Count) and then
                       Has_Element (Container, Position);
```

50/5
```
      procedure Insert_Child (Container : in out Tree;
                              Parent    : in     Cursor;
                              Before    : in     Cursor;
                              Position  :    out Cursor;
                              Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Parent /= No_Element
                         or else raise Constraint_Error) and then
                      (Meaningful_For (Container, Parent)
                         or else raise Program_Error) and then
                      (Meaningful_For (Container, Before)
                         or else raise Program_Error) and then
                      (Before = No_Element or else
                       Container.Parent (Before) = Parent
                         or else raise Constraint_Error),
              Post => (Node_Count (Container) =
                       Node_Count (Container)'Old + Count) and then
                       Has_Element (Container, Position);
```

51/5
```
      procedure Prepend_Child (Container : in out Tree;
                               Parent    : in     Cursor;
                               New_Item  : in     Element_Type;
                               Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Parent /= No_Element
                         or else raise Constraint_Error) and then
                      (Meaningful_For (Container, Parent)
                         or else raise Program_Error),
              Post => Node_Count (Container) =
                      Node_Count (Container)'Old + Count;
```

52/5
```
      procedure Append_Child (Container : in out Tree;
                              Parent    : in     Cursor;
                              New_Item  : in     Element_Type;
                              Count     : in     Count_Type := 1)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                         or else raise Program_Error) and then
                      (Parent /= No_Element
                         or else raise Constraint_Error) and then
                      (Meaningful_For (Container, Parent)
                         or else raise Program_Error),
              Post => Node_Count (Container) =
                      Node_Count (Container)'Old + Count;
```

```
procedure Delete_Children (Container : in out Tree;                    53/5
                           Parent    : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
        Post => (Node_Count (Container) = Node_Count (Container)'Old -
                    Child_Count (Container, Parent)'Old) and then
                  Child_Count (Container, Parent) = 0;
```

```
procedure Copy_Subtree (Target   : in out Tree;                        54/5
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (not Is_Root (Source)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) =
                Node_Count (Target)'Old + Subtree_Node_Count (Source),
        Global => in all;
```

```
procedure Copy_Local_Subtree (Target   : in out Tree;                  54.1/5
                              Parent   : in     Cursor;
                              Before   : in     Cursor;
                              Source   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Source)
                    or else raise Program_Error) and then
                 (not Is_Root (Source)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) = Node_Count (Target)'Old +
                  Subtree_Node_Count (Target, Source);
```

54.2/5
```
      procedure Copy_Subtree (Target   : in out Tree;
                              Parent   : in     Cursor;
                              Before   : in     Cursor;
                              Source   : in     Tree;
                              Subtree  : in     Cursor)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                          or else raise Program_Error) and then
                       (Parent /= No_Element
                          or else raise Constraint_Error) and then
                       (Meaningful_For (Target, Parent)
                          or else raise Program_Error) and then
                       (Meaningful_For (Target, Before)
                          or else raise Program_Error) and then
                       (Before = No_Element or else
                          Target.Parent (Before) = Parent
                          or else raise Constraint_Error) and then
                       (Meaningful_For (Source, Subtree)
                          or else raise Program_Error) and then
                       (not Is_Root (Source, Subtree)
                          or else raise Constraint_Error),
              Post => Node_Count (Target) = Node_Count (Target)'Old +
                        Subtree_Node_Count (Source, Subtree);
```

55/5
```
      procedure Splice_Subtree (Target   : in out Tree;
                                Parent   : in     Cursor;
                                Before   : in     Cursor;
                                Source   : in out Tree;
                                Position : in out Cursor)
         with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                          or else raise Program_Error) and then
                      (not Tampering_With_Cursors_Prohibited (Source)
                          or else raise Program_Error) and then
                      (Parent /= No_Element
                          or else raise Constraint_Error) and then
                      (Meaningful_For (Target, Parent)
                          or else raise Program_Error) and then
                      (Meaningful_For (Target, Before)
                          or else raise Program_Error) and then
                      (Before = No_Element or else
                       Target.Parent (Before) /= Parent
                          or else raise Constraint_Error) and then
                      (Position /= No_Element
                          or else raise Constraint_Error) and then
                      (Has_Element (Source, Position)
                          or else raise Program_Error) and then
                      (Target'Has_Same_Storage (Source) or else
                       Position = Before or else
                       Is_Ancestor_Of (Target, Position, Parent)
                          or else raise Constraint_Error),
              Post => (declare
                          Org_Sub_Count renames
                             Subtree_Node_Count (Source, Position)'Old;
                          Org_Target_Count renames Node_Count (Target)'Old;
                       begin
                         (if not Target'Has_Same_Storage (Source) then
                             Node_Count (Target) = Org_Target_Count +
                                Org_Sub_Count and then
                             Node_Count (Source) = Node_Count (Source)'Old -
                                Org_Sub_Count and then
                             Has_Element (Target, Position)
                          else
                             Target.Parent (Position) = Parent and then
                             Node_Count (Target) = Org_Target_Count));
```

```
procedure Splice_Subtree (Container: in out Tree;                                56/5
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Position : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Container, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Container.Parent (Before) /= Parent
                    or else raise Constraint_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error) and then
                 (Position = Before or else
                  Is_Ancestor_Of (Container, Position, Parent)
                    or else raise Constraint_Error),
        Post => (Node_Count (Container) =
                    Node_Count (Container)'Old and then
                  Container.Parent (Position) = Parent);

procedure Splice_Children (Target          : in out Tree;                        57/5
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source          : in out Tree;
                           Source_Parent   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error) and then
                 (Target_Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Target_Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Source_Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Source, Source_Parent)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Parent (Target, Before) /= Target_Parent
                    or else raise Constraint_Error) and then
                 (Target'Has_Same_Storage (Source) or else
                  Target_Parent = Source_Parent or else
                  Is_Ancestor_Of (Target, Source_Parent, Target_Parent)
                    or else raise Constraint_Error),
        Post => (declare
                    Org_Child_Count renames
                       Child_Count (Source, Source_Parent)'Old;
                    Org_Target_Count renames Node_Count (Target)'Old;
                 begin
                   (if not Target'Has_Same_Storage (Source) then
                       Node_Count (Target) = Org_Target_Count +
                          Org_Child_Count and then
                       Node_Count (Source) = Node_Count (Source)'Old -
                          Org_Child_Count
                    else
                       Node_Count (Target) = Org_Target_Count));
```

```
58/5        procedure Splice_Children (Container       : in out Tree;
                                       Target_Parent   : in     Cursor;
                                       Before          : in     Cursor;
                                       Source_Parent   : in     Cursor)
               with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                                or else raise Program_Error) and then
                             (Target_Parent /= No_Element
                                or else raise Constraint_Error) and then
                             (Meaningful_For (Container, Target_Parent)
                                or else raise Program_Error) and then
                             (Meaningful_For (Container, Before)
                                or else raise Program_Error) and then
                             (Source_Parent /= No_Element
                                or else raise Constraint_Error) and then
                             (Meaningful_For (Container, Source_Parent)
                                or else raise Program_Error) and then
                             (Before = No_Element or else
                              Parent (Container, Before) /= Target_Parent
                                or else raise Constraint_Error) and then
                             (Target_Parent = Source_Parent or else
                              Is_Ancestor_Of (Container, Source_Parent, Target_Parent)
                                or else raise Constraint_Error),
                    Post => Node_Count (Container) = Node_Count (Container)'Old;

59/5        function Parent (Position : Cursor) return Cursor
               with Nonblocking, Global => in all, Use_Formal => null,
                    Post => (if Position = No_Element or else
                                Is_Root (Position) then Parent'Result = No_Element);

59.1/5      function Parent (Container : Tree;
                             Position  : Cursor) return Cursor
               with Nonblocking, Global => null, Use_Formal => null,
                    Pre  => Meaningful_For (Container, Position)
                              or else raise Program_Error,
                    Post => (if Position = No_Element or else
                                Is_Root (Container, Position)
                                then Parent'Result = No_Element
                             else Has_Element (Container, Parent'Result));

60/5        function First_Child (Parent : Cursor) return Cursor
               with Nonblocking, Global => in all, Use_Formal => null,
                    Pre  => Parent /= No_Element or else raise Constraint_Error;

60.1/5      function First_Child (Container : Tree;
                                  Parent    : Cursor) return Cursor
               with Nonblocking, Global => null, Use_Formal => null,
                    Pre  => (Parent /= No_Element
                               or else raise Constraint_Error) and then
                            (Meaningful_For (Container, Parent)
                               or else raise Program_Error),
                    Post => First_Child'Result = No_Element or else
                            Has_Element (Container, First_Child'Result);

61/5        function First_Child_Element (Parent : Cursor) return Element_Type
               with Nonblocking, Global => in all, Use_Formal => Element_Type,
                    Pre  => (Parent /= No_Element and then
                             Last_Child (Parent) /= No_Element)
                                or else raise Constraint_Error;

61.1/5      function First_Child_Element (Container : Tree;
                                          Parent    : Cursor) return Element_Type
               with Nonblocking, Global => null, Use_Formal => Element_Type,
                    Pre  => (Parent /= No_Element
                               or else raise Constraint_Error) and then
                            (Meaningful_For (Container, Parent)
                               or else raise Program_Error) and then
                            (First_Child (Container, Parent) /= No_Element
                               or else raise Constraint_Error);
```

```
function Last_Child (Parent : Cursor) return Cursor                                    62/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Pre  => Parent /= No_Element or else raise Constraint_Error;

function Last_Child (Container : Tree;                                                 62.1/5
                     Parent    : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
        Post => Last_Child'Result = No_Element or else
                Has_Element (Container, Last_Child'Result);

function Last_Child_Element (Parent : Cursor) return Element_Type                      63/5
   with Nonblocking, Global => in all, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element and then
                 Last_Child (Parent) /= No_Element)
                    or else raise Constraint_Error;

function Last_Child_Element (Container : Tree;                                         63.1/5
                             Parent    : Cursor) return Element_Type
   with Nonblocking, Global => null, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error) and then
                 (Last_Child (Container, Parent) /= No_Element
                    or else raise Constraint_Error);

function Next_Sibling (Position : Cursor) return Cursor                                64/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                    then Next_Sibling'Result = No_Element);

function Next_Sibling (Container : Tree;                                               64.1/5
                       Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                    or else raise Program_Error,
        Post => (if Next_Sibling'Result = No_Element then
                    Position = No_Element or else
                    Is_Root (Container, Position) or else
                    Last_Child (Container, Parent (Container, Position))
                       = Position
                 else Has_Element (Container, Next_Sibling'Result));

procedure Next_Sibling (Position : in out Cursor)                                      65/5
   with Nonblocking, Global => in all, Use_Formal => null;

procedure Next_Sibling (Container : in     Tree;                                       65.1/5
                        Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                    or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));

function Previous_Sibling (Position : Cursor) return Cursor                            66/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                    then Previous_Sibling'Result = No_Element);
```

66.1/5
```
function Previous_Sibling (Container : Tree;
                           Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Previous_Sibling'Result = No_Element then
                   Position = No_Element or else
                   Is_Root (Container, Position) or else
                   First_Child (Container, Parent (Container, Position))
                      = Position
                 else Has_Element (Container, Previous_Sibling'Result));
```

67/5
```
procedure Previous_Sibling (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

67.1/5
```
procedure Previous_Sibling (Container : in     Tree;
                           Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

68/5
```
procedure Iterate_Children
     (Parent  : in Cursor;
      Process : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre    => Parent /= No_Element or else raise Constraint_Error,
        Global => in all, Use_Formal => null;
```

68.1/5
```
procedure Iterate_Children
     (Container : in Tree;
      Parent    : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error);
```

69/5
```
procedure Reverse_Iterate_Children
     (Parent  : in Cursor;
      Process : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre    => Parent /= No_Element or else raise Constraint_Error,
        Global => in all, Use_Formal => null;
```

69.1/5
```
procedure Reverse_Iterate_Children
     (Container : in Tree;
      Parent    : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error);
```

70/5
```
function Iterate_Children (Container : in Tree; Parent : in Cursor)
   return Tree_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

70.1/5
```
package Stable is
```

```
      type Tree (Base : not null access Multiway_Trees.Tree) is                    70.2/5
         tagged limited private
         with Constant_Indexing => Constant_Reference,
              Variable_Indexing => Reference,
              Default_Iterator  => Iterate,
              Iterator_Element  => Element_Type,
              Stable_Properties => (Node_Count),
              Global            => null,
              Default_Initial_Condition => Node_Count (Tree) = 1,
              Preelaborable_Initialization;

      type Cursor is private                                                       70.3/5
         with Preelaborable_Initialization;

      Empty_Tree : constant Tree;                                                  70.4/5

      No_Element : constant Cursor;                                                70.5/5

      function Has_Element (Position : Cursor) return Boolean                      70.6/5
         with Nonblocking, Global => in all, Use_Formal => null;

      package Tree_Iterator_Interfaces is new                                      70.7/5
         Ada.Iterator_Interfaces (Cursor, Has_Element);

      procedure Assign (Target : in out Multiway_Trees.Tree;                       70.8/5
                        Source : in Tree)
         with Post => Node_Count (Source) = Node_Count (Target);

      function Copy (Source : Multiway_Trees.Tree) return Tree                     70.9/5
         with Post => Node_Count (Copy'Result) = Node_Count (Source);

      type Constant_Reference_Type                                                 70.10/6
            (Element : not null access constant Element_Type) is
              limited private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null,
              Default_Initial_Condition => (raise Program_Error);

      type Reference_Type                                                          70.11/6
            (Element : not null access Element_Type) is limited private
         with Implicit_Dereference => Element,
              Nonblocking, Global => null,
              Default_Initial_Condition => (raise Program_Error);

      -- Additional subprograms as described in the text                         70.12/5
      -- are declared here.

   private                                                                         70.13/5

      ... -- not specified by the language                                        70.14/5

   end Stable;                                                                    70.15/5

private                                                                           71/3
   ... -- not specified by the language
end Ada.Containers.Multiway_Trees;
```

The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the functions Find, Reverse_Find, Equal_Subtree, and "=" on tree values return an unspecified value. The exact arguments and number of calls of this generic formal function by the functions Find, Reverse_Find, Equal_Subtree, and "=" on tree values are unspecified. 72/3

The type Tree is used to represent trees. The type Tree needs finalization (see 7.6). 73/3

Empty_Tree represents the empty Tree object. It contains only the root node (Node_Count (Empty_Tree) returns 1). If an object of type Tree is not otherwise initialized, it is initialized to the same value as Empty_Tree. 74/3

75/3 No_Element represents a cursor that designates no element. If an object of type Cursor is not otherwise initialized, it is initialized to the same value as No_Element.

76/5 The primitive "=" operator for type Cursor returns True if both cursors are No_Element, or designate the same element in the same container.

77/3 Execution of the default implementation of the Input, Output, Read, or Write attribute of type Cursor raises Program_Error.

78/5 Tree'Write for a Tree object *T* writes Node_Count(*T*) - 1 elements of the tree to the stream. It may also write additional information about the tree.

79/3 Tree'Read reads the representation of a tree from the stream, and assigns to *Item* a tree with the same elements and structure as was written by Tree'Write.

80/5 Some operations check for “tampering with cursors” of a container because they depend on the set of elements of the container remaining constant, and others check for “tampering with elements” of a container because they depend on elements of the container not being replaced. When tampering with cursors is *prohibited* for a particular tree object *T*, Program_Error is propagated by the finalization of *T*, as well as by a call that passes *T* to certain of the operations of this package, as indicated by the precondition of such an operation. Similarly, when tampering with elements is *prohibited* for *T*, Program_Error is propagated by a call that passes *T* to certain of the other operations of this package, as indicated by the precondition of such an operation.

Paragraphs 81 through 90 are removed as preconditions now describe these rules.

91/5
```
function Has_Element (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

92/3 Returns True if Position designates an element, and returns False otherwise. In particular, Has_Element returns False if the cursor designates a root node or equals No_Element.

92.1/5
```
function Has_Element (Container : Tree; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

92.2/5 Returns True if Position designates an element in Container, and returns False otherwise. In particular, Has_Element returns False if the cursor designates a root node or equals No_Element.

93/3
```
function Equal_Subtree (Left_Position : Cursor;
                        Right_Position: Cursor) return Boolean;
```

94/3 If Left_Position or Right_Position equals No_Element, propagates Constraint_Error. If the number of child nodes of the element designated by Left_Position is different from the number of child nodes of the element designated by Right_Position, the function returns False. If Left_Position designates a root node and Right_Position does not, the function returns False. If Right_Position designates a root node and Left_Position does not, the function returns False. Unless both cursors designate a root node, the elements are compared using the generic formal equality operator. If the result of the element comparison is False, the function returns False. Otherwise, it calls Equal_Subtree on a cursor designating each child element of the element designated by Left_Position and a cursor designating the corresponding child element of the element designated by Right_Position. If any such call returns False, the function returns False; otherwise, it returns True. Any exception raised during the evaluation of element equality is propagated.

```
function "=" (Left, Right : Tree) return Boolean;
```
95/3

If Left and Right denote the same tree object, then the function returns True. Otherwise, it calls Equal_Subtree with cursors designating the root nodes of Left and Right; the result is returned. Any exception raised during the evaluation of Equal_Subtree is propagated. 96/3

```
function Tampering_With_Cursors_Prohibited
   (Container : Tree) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
96.1/5

Returns True if tampering with cursors or tampering with elements is currently prohibited for Container, and returns False otherwise. 96.2/5

```
function Tampering_With_Elements_Prohibited
   (Container : Tree) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
96.3/5

Always returns False, regardless of whether tampering with elements is prohibited. 96.4/5

```
function Is_Empty (Container : Tree) return Boolean
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Is_Empty'Result = (Node_Count (Container) = 1);
```
96.5/5

Returns True if Container is empty. 96.6/5

```
function Node_Count (Container : Tree) return Count_Type
   with Nonblocking, Global => null, Use_Formal => null;
```
97/5

Node_Count returns the number of nodes in Container. 98/3

```
function Subtree_Node_Count (Position : Cursor) return Count_Type
   with Nonblocking, Global => in all, Use_Formal => null);
```
99/5

If Position is No_Element, Subtree_Node_Count returns 0; otherwise, Subtree_Node_Count returns the number of nodes in the subtree that is rooted by Position. 100/3

```
function Subtree_Node_Count (Container : Tree; Position : Cursor)
   return Count_Type
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;
```
101/5

If Position is No_Element, Subtree_Node_Count returns 0; otherwise, Subtree_Node_Count returns the number of nodes in the subtree of Container that is rooted by Position. 102/5

```
function Depth (Position : Cursor) return Count_Type
   with Nonblocking, Global => in all, Use_Formal => null;
```
103/5

If Position equals No_Element, Depth returns 0; otherwise, Depth returns the number of ancestor nodes of the node designated by Position (including the node itself). 104/3

```
function Depth (Container : Tree; Position : Cursor)
   return Count_Type
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;
```
104.1/5

If Position equals No_Element, Depth returns 0; otherwise, Depth returns the number of ancestor nodes of the node of Container designated by Position (including the node itself). 104.2/5

105/5
```
function Is_Root (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

106/3 Is_Root returns True if the Position designates the root node of some tree; and returns False otherwise.

106.1/5
```
function Is_Root (Container : Tree; Position : Cursor)
   return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```

106.2/5 Is_Root returns True if the Position designates the root node of Container; and returns False otherwise.

107/5
```
function Is_Leaf (Position : Cursor) return Boolean
   with Nonblocking, Global => in all, Use_Formal => null;
```

108/3 Is_Leaf returns True if Position designates a node that does not have any child nodes; and returns False otherwise.

108.1/5
```
function Is_Leaf (Container : Tree; Position : Cursor)
   return Boolean
   with Pre => Meaningful_For (Container, Position)
                  or else raise Program_Error,
        Nonblocking, Global => null, Use_Formal => null;
```

108.2/5 Is_Leaf returns True if Position designates a node in Container that does not have any child nodes; and returns False otherwise.

108.3/5
```
function Is_Ancestor_Of (Container : Tree;
                         Parent   : Cursor;
                         Position : Cursor) return Boolean
   with Pre => (Meaningful_For (Container, Position)
                  or else raise Program_Error) and then
               (Meaningful_For (Container, Parent)
                  or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => null;
```

108.4/5 Is_Ancestor_Of returns True if Parent designates an ancestor node of Position (including Position itself), and returns False otherwise.

109/5
```
function Root (Container : Tree) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Post => Root'Result /= No_Element and then
                not Has_Element (Container, Root'Result);
```

110/3 Root returns a cursor that designates the root node of Container.

111/5
```
procedure Clear (Container : in out Tree)
   with Pre  => not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error,
        Post => Node_Count (Container) = 1;
```

112/3 Removes all the elements from Container.

113/5
```
function Element (Position : Cursor) return Element_Type
   with Pre => (Position /= No_Element or else
                  raise Constraint_Error) and then
               (Has_Element (Position) or else raise Program_Error),
        Nonblocking, Global => in all, Use_Formal => Element_Type;
```

114/5 Element returns the element designated by Position.

```
function Element (Container : Tree;                                    114.1/5
                  Position  : Cursor) return Element_Type
   with Pre => (Position /= No_Element
                   or else raise Constraint_Error) and then
               (Has_Element (Container, Position)
                   or else raise Program_Error),
        Nonblocking, Global => null, Use_Formal => Element_Type;
```

Element returns the element designated by Position in Container. 114.2/5

```
procedure Replace_Element (Container : in out Tree;                    115/5
                           Position  : in     Cursor;
                           New_item  : in     Element_Type)
   with Pre  => (not Tampering_With_Elements_Prohibited (Container)
                   or else raise Program_Error) and then
                (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

Replace_Element assigns the value New_Item to the element designated by Position. For the purposes of determining whether the parameters overlap in a call to Replace_Element, the Container parameter is not considered to overlap with any object (including itself). 116/5

```
procedure Query_Element                                                117/5
  (Position : in Cursor;
   Process  : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Position)
                   or else raise Program_Error),
     Global => in all;
```

Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of the tree that contains the element designated by Position is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 118/5

```
procedure Query_Element                                                118.1/5
  (Container : in Tree;
   Position  : in Cursor;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

Query_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 118.2/5

```
procedure Update_Element                                               119/5
  (Container : in out Tree;
   Position  : in     Cursor;
   Process   : not null access procedure
                  (Element : in out Element_Type))
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                   or else raise Program_Error);
```

Update_Element calls Process.**all** with the element designated by Position as the argument. Tampering with the elements of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 120/5

121/3 If Element_Type is unconstrained and definite, then the actual Element parameter of Process.**all** shall be unconstrained.

122/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

123/6
```
type Reference_Type
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

124/3 The types Constant_Reference_Type and Reference_Type need finalization.

125/5 *This paragraph was deleted.*

126/5
```
function Constant_Reference (Container : aliased in Tree;
                             Position  : in Cursor)
   return Constant_Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

127/3 This function (combined with the Constant_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read access to an individual element of a tree given a cursor.

128/5 Constant_Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

129/5
```
function Reference (Container : aliased in out Tree;
                    Position  : in Cursor)
   return Reference_Type
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post   => Tampering_With_Cursors_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

130/3 This function (combined with the Variable_Indexing and Implicit_Dereference aspects) provides a convenient way to gain read and write access to an individual element of a tree given a cursor.

131/5 Reference returns an object whose discriminant is an access value that designates the element designated by Position. Tampering with the elements of Container is prohibited while the object returned by Reference exists and has not been finalized.

132/5
```
procedure Assign (Target : in out Tree; Source : in Tree)
   with Pre  => not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error,
        Post => Node_Count (Source) = Node_Count (Target);
```

133/3 If Target denotes the same object as Source, the operation has no effect. Otherwise, the elements of Source are copied to Target as for an assignment_statement assigning Source to Target.

```
function Copy (Source : Tree) return Tree                                    134/5
   with Post =>
           Node_Count (Copy'Result) = Node_Count (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result);
```

Returns a tree with the same structure as Source and whose elements are initialized from the corresponding elements of Source. 135/3

```
procedure Move (Target : in out Tree;                                        136/5
                Source : in out Tree)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_Cursors_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                   Node_Count (Target) = Node_Count (Source'Old) and then
                   Node_Count (Source) = 1);
```

If Target denotes the same object as Source, then the operation has no effect. Otherwise, Move first calls Clear (Target). Then, the nodes other than the root node in Source are moved to Target (in the same positions). After Move completes, Node_Count (Target) is the number of nodes originally in Source, and Node_Count (Source) is 1. 137/3

```
procedure Delete_Leaf (Container : in out Tree;                              138/5
                       Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error) and then
                 (Is_Leaf (Container, Position)
                    or else raise Constraint_Error),
        Post =>
           Node_Count (Container)'Old = Node_Count (Container) + 1 and then
           Position = No_Element;
```

Delete_Leaf removes (from Container) the element designated by Position, and Position is set to No_Element. 139/5

```
procedure Delete_Subtree (Container : in out Tree;                           140/5
                          Position  : in out Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error),
        Post => Node_Count (Container)'Old = Node_Count (Container) +
                   Subtree_Node_Count (Container, Position)'Old and then
                 Position = No_Element;
```

Delete_Subtree removes (from Container) the subtree designated by Position (that is, all descendants of the node designated by Position including the node itself), and Position is set to No_Element. 141/5

142/5
```
procedure Swap (Container : in out Tree;
                I, J      : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (I /= No_Element or else Constraint_Error) and then
                (J /= No_Element or else Constraint_Error) and then
                (Has_Element (Container, I)
                   or else raise Program_Error) and then
                (Has_Element (Container, J)
                   or else raise Program_Error);
```

143/5 Swap exchanges the values of the elements designated by I and J.

144/5
```
function Find (Container : Tree;
               Item      : Element_Type)
   return Cursor
   with Post => (if Find'Result /= No_Element
                 then Has_Element (Container, Find'Result));
```

145/3 Find searches the elements of Container for an element equal to Item (using the generic formal equality operator). The search starts at the root node. The search traverses the tree in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

146/5
```
function Find_In_Subtree (Position : Cursor;
                          Item     : Element_Type)
   return Cursor
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Post => (if Find_In_Subtree'Result = No_Element
                 then Has_Element (Find_In_Subtree'Result)),
        Global => in all;
```

147/5 Find_In_Subtree searches the subtree rooted by Position for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position. The search traverses the subtree in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

147.1/5
```
function Find_In_Subtree (Container : Tree;
                          Position  : Cursor;
                          Item      : Element_Type)
   return Cursor
   with Pre  => (Position /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                   or else raise Program_Error),
        Post => (if Find_In_Subtree'Result = No_Element
                 then Has_Element (Container, Find_In_Subtree'Result));
```

147.2/5 Find_In_Subtree searches the subtree of Container rooted by Position for an element equal to Item (using the generic formal equality operator). The search starts at the element designated by Position. The search traverses the subtree in a depth-first order. If no equal element is found, then Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered.

```
function Ancestor_Find (Position : Cursor;                                  148/5
                        Item     : Element_Type)
   return Cursor
   with Pre  => Position /= No_Element or else raise Constraint_Error,
        Post => (if Ancestor_Find'Result = No_Element
                 then Has_Element (Container, Ancestor_Find'Result)),
        Global => in all;
```

Ancestor_Find searches for an element equal to Item (using the generic formal equality operator). The search starts at the node designated by Position, and checks each ancestor proceeding toward the root of the subtree. If no equal element is found, then Ancestor_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 149/5

```
function Ancestor_Find (Container : Tree;                                   149.1/5
                        Position  : Cursor;
                        Item      : Element_Type)
   return Cursor
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                    or else raise Program_Error),
        Post => (if Ancestor_Find'Result = No_Element
                 then Has_Element (Container, Ancestor_Find'Result));
```

Ancestor_Find searches for an element equal to Item (using the generic formal equality operator). The search starts at the node designated by Position in Container, and checks each ancestor proceeding toward the root of the subtree. If no equal element is found, then Ancestor_Find returns No_Element. Otherwise, it returns a cursor designating the first equal element encountered. 149.2/5

```
function Contains (Container : Tree;                                        150/3
                   Item      : Element_Type) return Boolean;
```

Equivalent to Find (Container, Item) /= No_Element. 151/3

```
procedure Iterate                                                           152/5
  (Container : in Tree;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit;
```

Iterate calls Process.**all** with a cursor that designates each element in Container, starting from the root node and proceeding in a depth-first order. Tampering with the cursors of Container is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 153/4

```
procedure Iterate_Subtree                                                   154/5
  (Position  : in Cursor;
   Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

Iterate_Subtree calls Process.**all** with a cursor that designates each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 155/5

155.1/5
```
procedure Iterate_Subtree
   (Container : in Tree;
    Position  : in Cursor;
    Process   : not null access procedure (Position : in Cursor))
    with Allows_Exit,
         Pre  => (Position /= No_Element
                     or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Position)
                     or else raise Program_Error);
```

155.2/5 Iterate_Subtree calls Process.**all** with a cursor that designates each element in the subtree rooted by the node designated by Position in Container, starting from the node designated by Position and proceeding in a depth-first order. Tampering with the cursors of the tree that contains the element designated by Position is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

156/5
```
function Iterate (Container : in Tree)
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Post => Tampering_With_Cursors_Prohibited (Container);
```

157/5 Iterate returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in Container, starting from the root node and proceeding in a depth-first order when used as a forward iterator, and processing all nodes concurrently when used as a parallel iterator. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

158/5
```
function Iterate_Subtree (Position : in Cursor)
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Pre    => Position /= No_Element or else raise Constraint_Error,
        Global => in all;
```

159/5 Iterate_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position, starting from the node designated by Position and proceeding in a depth-first order when used as a forward iterator, and processing all nodes in the subtree concurrently when used as a parallel iterator. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

159.1/5
```
function Iterate_Subtree (Container : in Tree; Position : in Cursor)
   return Tree_Iterator_Interfaces.Parallel_Iterator'Class
   with Pre  => (Position /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Position)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

159.2/5 Iterate_Subtree returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each element in the subtree rooted by the node designated by Position in Container, starting from the node designated by Position and proceeding in a depth-first order when used as a forward iterator, and processing all nodes in the subtree concurrently when used as a parallel iterator. Tampering with the cursors of the container that contains the node designated by Position is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization.

```
function Child_Count (Parent : Cursor) return Count_Type                    160/5
   with Post => (if Parent = No_Element then Child_Count'Result = 0),
         Nonblocking, Global => in all, Use_Formal => null;
```

Child_Count returns the number of child nodes of the node designated by Parent. 161/3

```
function Child_Count (Container : Tree; Parent : Cursor)                    161.1/5
   return Count_Type
   with Pre  => Meaningful_For (Container, Parent)
                   or else raise Program_Error,
        Post => (if Parent = No_Element then Child_Count'Result = 0),
        Nonblocking, Global => null, Use_Formal => null;
```

Child_Count returns the number of child nodes of the node designated by Parent in Container. 161.2/5

```
function Child_Depth (Parent, Child : Cursor) return Count_Type             162/5
   with Pre  => (Parent /= No_Element and then Child /= No_Element)
                   or else raise Constraint_Error,
     Nonblocking, Global => in all, Use_Formal => null;
```

Child_Depth returns the number of ancestor nodes of Child (including Child itself), up to but not including Parent; Program_Error is propagated if Parent is not an ancestor of Child. 163/5

```
function Child_Depth (Container : Tree; Parent, Child : Cursor)             163.1/5
   return Count_Type
   with Pre  => ((Parent /= No_Element and then Child /= No_Element)
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                (Meaningful_For (Container, Child)
                   or else raise Program_Error),
          Nonblocking, Global => null, Use_Formal => null;
```

Child_Depth returns the number of ancestor nodes of Child within Container (including Child itself), up to but not including Parent; Program_Error is propagated if Parent is not an ancestor of Child. 163.2/5

```
procedure Insert_Child (Container : in out Tree;                            164/5
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                   or else raise Program_Error) and then
                (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                (Meaningful_For (Container, Before)
                   or else raise Program_Error) and then
                (Before = No_Element or else
                 Container.Parent (Before) = Parent
                   or else raise Constraint_Error),
        Post => Node_Count (Container) =
                Node_Count (Container)'Old + Count;
```

Insert_Child allocates Count nodes containing copies of New_Item and inserts them as children of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Any exception raised during allocation of internal storage is propagated, and Container is not modified. 165/5

166/5
```
   procedure Insert_Child (Container : in out Tree;
                           Parent    : in     Cursor;
                           Before    : in     Cursor;
                           New_Item  : in     Element_Type;
                           Position  :    out Cursor;
                           Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Parent /= No_Element
                      or else raise Constraint_Error) and then
                   (Meaningful_For (Container, Parent)
                      or else raise Program_Error) and then
                   (Meaningful_For (Container, Before)
                      or else raise Program_Error) and then
                   (Before = No_Element or else
                    Container.Parent (Before) = Parent
                      or else raise Constraint_Error),
           Post => (Node_Count (Container) =
                    Node_Count (Container)'Old + Count) and then
                    Has_Element (Container, Position);
```

167/5 Insert_Child allocates Count nodes containing copies of New_Item and inserts them as children of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Position designates the first newly-inserted node, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

168/5
```
   procedure Insert_Child (Container : in out Tree;
                           Parent    : in     Cursor;
                           Before    : in     Cursor;
                           Position  :    out Cursor;
                           Count     : in     Count_Type := 1)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                      or else raise Program_Error) and then
                   (Parent /= No_Element
                      or else raise Constraint_Error) and then
                   (Meaningful_For (Container, Parent)
                      or else raise Program_Error) and then
                   (Meaningful_For (Container, Before)
                      or else raise Program_Error) and then
                   (Before = No_Element or else
                    Container.Parent (Before) = Parent
                      or else raise Constraint_Error),
           Post => (Node_Count (Container) =
                    Node_Count (Container)'Old + Count) and then
                    Has_Element (Container, Position);
```

169/5 Insert_Child allocates Count nodes, the elements contained in the new nodes are initialized by default (see 3.3.1), and the new nodes are inserted as children of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. Position designates the first newly-inserted node, or if Count equals 0, then Position is assigned the value of Before. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Prepend_Child (Container : in out Tree;                                    170/5
                         Parent    : in     Cursor;
                         New_Item  : in     Element_Type;
                         Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
        Post => Node_Count (Container) =
                Node_Count (Container)'Old + Count;
```

Equivalent to Insert_Child (Container, Parent, First_Child (Container, Parent), New_Item, Count). 171/3

```
procedure Append_Child (Container : in out Tree;                                     172/5
                        Parent    : in     Cursor;
                        New_Item  : in     Element_Type;
                        Count     : in     Count_Type := 1)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
        Post => Node_Count (Container) =
                Node_Count (Container)'Old + Count;
```

Equivalent to Insert_Child (Container, Parent, No_Element, New_Item, Count). 173/3

```
procedure Delete_Children (Container : in out Tree;                                  174/5
                           Parent    : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
        Post => (Node_Count (Container) = Node_Count (Container)'Old -
                   Child_Count (Container, Parent)'Old) and then
                  Child_Count (Container, Parent) = 0;
```

Delete_Children removes (from Container) all of the descendants of Parent other than Parent itself. 175/5

176/5
```
procedure Copy_Subtree (Target   : in out Tree;
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (not Is_Root (Source)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) =
                Node_Count (Target)'Old + Subtree_Node_Count (Source),
        Global => in all;
```

177/5 If Source is equal to No_Element, then the operation has no effect. Otherwise, the subtree rooted by Source (which can be from any tree; it does not have to be a subtree of Target) is copied (new nodes are allocated to create a new subtree with the same structure as the Source subtree, with each element initialized from the corresponding element of the Source subtree) and inserted into Target as a child of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. The parent of the newly created subtree is set to Parent, and the overall count of Target is incremented by Subtree_Node_Count (Source). Any exception raised during allocation of internal storage is propagated, and Container is not modified.

177.1/5
```
procedure Copy_Local_Subtree (Target   : in out Tree;
                              Parent   : in     Cursor;
                              Before   : in     Cursor;
                              Source   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Source)
                    or else raise Program_Error) and then
                 (not Is_Root (Source)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) = Node_Count (Target)'Old +
                  Subtree_Node_Count (Target, Source);
```

177.2/5 If Source is equal to No_Element, then the operation has no effect. Otherwise, the subtree rooted by Source in Target is copied (new nodes are allocated to create a new subtree with the same structure as the Source subtree, with each element initialized from the corresponding element of the Source subtree) and inserted into Target as a child of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. The

parent of the newly created subtree is set to Parent. Any exception raised during allocation of internal storage is propagated, and Container is not modified.

```
procedure Copy_Subtree (Target   : in out Tree;                                    177.3/5
                        Parent   : in     Cursor;
                        Before   : in     Cursor;
                        Source   : in     Tree;
                        Subtree  : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Target, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Target, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                    Target.Parent (Before) = Parent
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Source, Subtree)
                    or else raise Program_Error) and then
                 (not Is_Root (Source, Subtree)
                    or else raise Constraint_Error),
        Post => Node_Count (Target) = Node_Count (Target)'Old +
                  Subtree_Node_Count (Source, Subtree);
```

If Subtree is equal to No_Element, then the operation has no effect. Otherwise, the subtree rooted by Subtree in Source is copied (new nodes are allocated to create a new subtree with the same structure as the Subtree, with each element initialized from the corresponding element of the Subtree) and inserted into Target as a child of Parent. If Parent already has child nodes, then the new nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the new nodes are inserted after the last existing child node of Parent. The parent of the newly created subtree is set to Parent. Any exception raised during allocation of internal storage is propagated, and Container is not modified. 177.4/5

178/5
```
   procedure Splice_Subtree (Target   : in out Tree;
                             Parent   : in     Cursor;
                             Before   : in     Cursor;
                             Source   : in out Tree;
                             Position : in out Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                       or else raise Program_Error) and then
                   (Parent /= No_Element
                       or else raise Constraint_Error) and then
                   (Meaningful_For (Target, Parent)
                       or else raise Program_Error) and then
                   (Meaningful_For (Target, Before)
                       or else raise Program_Error) and then
                   (Before = No_Element or else
                    Target.Parent (Before) /= Parent
                       or else raise Constraint_Error) and then
                   (Position /= No_Element
                       or else raise Constraint_Error) and then
                   (Has_Element (Source, Position)
                       or else raise Program_Error) and then
                   (Target'Has_Same_Storage (Source) or else
                    Position = Before or else
                    Is_Ancestor_Of (Target, Position, Parent)
                       or else raise Constraint_Error),
           Post => (declare
                       Org_Sub_Count renames
                           Subtree_Node_Count (Source, Position)'Old;
                       Org_Target_Count renames Node_Count (Target)'Old;
                    begin
                      (if not Target'Has_Same_Storage (Source) then
                          Node_Count (Target) = Org_Target_Count +
                             Org_Sub_Count and then
                          Node_Count (Source) = Node_Count (Source)'Old -
                             Org_Sub_Count and then
                          Has_Element (Target, Position)
                       else
                          Target.Parent (Position) = Parent and then
                          Node_Count (Target) = Org_Target_Count));
```

179/5 If Source denotes the same object as Target, then: if Position equals Before there is no effect; otherwise, the subtree rooted by the element designated by Position is moved to be a child of Parent. If Parent already has child nodes, then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. In each of these cases, Position and the count of Target are unchanged, and the parent of the element designated by Position is set to Parent.

180/3 Otherwise (if Source does not denote the same object as Target), the subtree designated by Position is removed from Source and moved to Target. The subtree is inserted as a child of Parent. If Parent already has child nodes, then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. In each of these cases, the count of Target is incremented by Subtree_Node_Count (Position), and the count of Source is decremented by Subtree_Node_Count (Position), Position is updated to represent an element in Target.

```
procedure Splice_Subtree (Container: in out Tree;                                    181/5
                          Parent   : in     Cursor;
                          Before   : in     Cursor;
                          Position : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Container, Before)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Container.Parent (Before) /= Parent
                    or else raise Constraint_Error) and then
                 (Position /= No_Element
                    or else raise Constraint_Error) and then
                 (Has_Element (Container, Position)
                    or else raise Program_Error) and then
                 (Position = Before or else
                  Is_Ancestor_Of (Container, Position, Parent)
                    or else raise Constraint_Error),
        Post => (Node_Count (Container) =
                    Node_Count (Container)'Old and then
                  Container.Parent (Position) = Parent);
```

If Position equals Before, there is no effect. Otherwise, the subtree rooted by the element designated by Position is moved to be a child of Parent. If Parent already has child nodes, then the moved nodes are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved nodes are inserted after the last existing child node of Parent. The parent of the element designated by Position is set to Parent. 182/5

183/5
```
   procedure Splice_Children (Target          : in out Tree;
                              Target_Parent   : in     Cursor;
                              Before          : in     Cursor;
                              Source          : in out Tree;
                              Source_Parent   : in     Cursor)
      with Pre  => (not Tampering_With_Cursors_Prohibited (Target)
                       or else raise Program_Error) and then
                   (not Tampering_With_Cursors_Prohibited (Source)
                       or else raise Program_Error) and then
                   (Target_Parent /= No_Element
                       or else raise Constraint_Error) and then
                   (Meaningful_For (Target, Target_Parent)
                       or else raise Program_Error) and then
                   (Meaningful_For (Target, Before)
                       or else raise Program_Error) and then
                   (Source_Parent /= No_Element
                       or else raise Constraint_Error) and then
                   (Meaningful_For (Source, Source_Parent)
                       or else raise Program_Error) and then
                   (Before = No_Element or else
                    Parent (Target, Before) /= Target_Parent
                       or else raise Constraint_Error) and then
                   (Target'Has_Same_Storage (Source) or else
                    Target_Parent = Source_Parent or else
                    Is_Ancestor_Of (Target, Source_Parent, Target_Parent)
                       or else raise Constraint_Error),
           Post => (declare
                       Org_Child_Count renames
                          Child_Count (Source, Source_Parent)'Old;
                       Org_Target_Count renames Node_Count (Target)'Old;
                    begin
                      (if not Target'Has_Same_Storage (Source) then
                          Node_Count (Target) = Org_Target_Count +
                             Org_Child_Count and then
                          Node_Count (Source) = Node_Count (Source)'Old -
                             Org_Child_Count
                       else
                          Node_Count (Target) = Org_Target_Count));
```

184/5 *This paragraph was deleted.*

185/3 If Source denotes the same object as Target, then:

186/3 - if Target_Parent equals Source_Parent there is no effect; else

187/5 - *This paragraph was deleted.*

188/3 - the child elements (and the further descendants) of Source_Parent are moved to be child elements of Target_Parent. If Target_Parent already has child elements, then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. The parent of each moved child element is set to Target_Parent.

189/3 Otherwise (if Source does not denote the same object as Target), the child elements (and the further descendants) of Source_Parent are removed from Source and moved to Target. The child elements are inserted as children of Target_Parent. If Target_Parent already has child elements, then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. In each of these cases, the overall count of Target is incremented by Subtree_Node_Count (Source_Parent)-1, and the overall count of Source is decremented by Subtree_Node_Count (Source_Parent)-1.

```
procedure Splice_Children (Container       : in out Tree;                    190/5
                           Target_Parent   : in     Cursor;
                           Before          : in     Cursor;
                           Source_Parent   : in     Cursor)
   with Pre  => (not Tampering_With_Cursors_Prohibited (Container)
                    or else raise Program_Error) and then
                 (Target_Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Target_Parent)
                    or else raise Program_Error) and then
                 (Meaningful_For (Container, Before)
                    or else raise Program_Error) and then
                 (Source_Parent /= No_Element
                    or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Source_Parent)
                    or else raise Program_Error) and then
                 (Before = No_Element or else
                  Parent (Container, Before) /= Target_Parent
                    or else raise Constraint_Error) and then
                 (Target_Parent = Source_Parent or else
                  Is_Ancestor_Of (Container, Source_Parent, Target_Parent)
                    or else raise Constraint_Error),
        Post => Node_Count (Container) = Node_Count (Container)'Old;
```

If Target_Parent equals Source_Parent there is no effect. Otherwise, the child elements (and the further descendants) of Source_Parent are moved to be child elements of Target_Parent. If Target_Parent already has child elements, then the moved elements are inserted prior to the node designated by Before, or, if Before equals No_Element, the moved elements are inserted after the last existing child node of Target_Parent. The parent of each moved child element is set to Target_Parent. 191/5

```
function Parent (Position : Cursor) return Cursor                            192/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element or else
                    Is_Root (Position) then Parent'Result = No_Element);
```

Returns a cursor designating the parent node of the node designated by Position. 193/5

```
function Parent (Container : Tree;                                           193.1/5
                 Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                    or else raise Program_Error,
        Post => (if Position = No_Element or else
                   Is_Root (Container, Position)
                   then Parent'Result = No_Element
                 else Has_Element (Container, Parent'Result));
```

Returns a cursor designating the parent node of the node designated by Position in Container. 193.2/5

```
function First_Child (Parent : Cursor) return Cursor                         194/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Pre  => Parent /= No_Element or else raise Constraint_Error;
```

First_Child returns a cursor designating the first child node of the node designated by Parent; if there is no such node, No_Element is returned. 195/5

195.1/5
```
function First_Child (Container : Tree;
                      Parent    : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => First_Child'Result = No_Element or else
                Has_Element (Container, First_Child'Result);
```

195.2/5 First_Child returns a cursor designating the first child node of the node designated by Parent in Container; if there is no such node, No_Element is returned.

196/5
```
function First_Child_Element (Parent : Cursor) return Element_Type
   with Nonblocking, Global => in all, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element and then
                 Last_Child (Parent) /= No_Element)
                     or else raise Constraint_Error;
```

197/3 Equivalent to Element (First_Child (Parent)).

197.1/5
```
function First_Child_Element (Container : Tree;
                              Parent    : Cursor) return Element_Type
   with Nonblocking, Global => null, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                (First_Child (Container, Parent) /= No_Element
                   or else raise Constraint_Error);
```

197.2/5 Equivalent to Element (Container, First_Child (Container, Parent)).

198/5
```
function Last_Child (Parent : Cursor) return Cursor
   with Nonblocking, Global => in all, Use_Formal => null,
        Pre  => Parent /= No_Element or else raise Constraint_Error;
```

199/5 Last_Child returns a cursor designating the last child node of the node designated by Parent; if there is no such node, No_Element is returned.

199.1/5
```
function Last_Child (Container : Tree;
                     Parent    : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error),
        Post => Last_Child'Result = No_Element or else
                Has_Element (Container, Last_Child'Result);
```

199.2/5 Last_Child returns a cursor designating the last child node of the node designated by Parent in Container; if there is no such node, No_Element is returned.

200/5
```
function Last_Child_Element (Parent : Cursor) return Element_Type
   with Nonblocking, Global => in all, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element and then
                 Last_Child (Parent) /= No_Element)
                     or else raise Constraint_Error;
```

201/3 Equivalent to Element (Last_Child (Parent)).

```
function Last_Child_Element (Container : Tree;                                    201.1/5
                             Parent    : Cursor) return Element_Type
   with Nonblocking, Global => null, Use_Formal => Element_Type,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                 (Meaningful_For (Container, Parent)
                   or else raise Program_Error) and then
                 (Last_Child (Container, Parent) /= No_Element
                   or else raise Constraint_Error);
```

Equivalent to Element (Container, Last_Child (Container, Parent)). 201.2/5

```
function Next_Sibling (Position : Cursor) return Cursor                           202/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                   then Next_Sibling'Result = No_Element);
```

If Position equals No_Element or designates the last child node of its parent, then Next_Sibling returns the value No_Element. Otherwise, it returns a cursor that designates the successor (with the same parent) of the node designated by Position. 203/3

```
function Next_Sibling (Container : Tree;                                          203.1/5
                       Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Next_Sibling'Result = No_Element then
                   Position = No_Element or else
                   Is_Root (Container, Position) or else
                   Last_Child (Container, Parent (Container, Position))
                     = Position
                 else Has_Element (Container, Next_Sibling'Result));
```

Next_Sibling returns a cursor that designates the successor (with the same parent) of the node designated by Position in Container. 203.2/5

```
function Previous_Sibling (Position : in out Cursor)                              204/5
   with Nonblocking, Global => in all, Use_Formal => null,
        Post => (if Position = No_Element
                   then Previous_Sibling'Result = No_Element);
```

If Position equals No_Element or designates the first child node of its parent, then Previous_Sibling returns the value No_Element. Otherwise, it returns a cursor that designates the predecessor (with the same parent) of the node designated by Position. 205/3

```
function Previous_Sibling (Container : Tree;                                      205.1/5
                           Position  : Cursor) return Cursor
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Previous_Sibling'Result = No_Element then
                   Position = No_Element or else
                   Is_Root (Container, Position) or else
                   First_Child (Container, Parent (Container, Position))
                     = Position
                 else Has_Element (Container, Previous_Sibling'Result));
```

Previous_Sibling returns a cursor that designates the predecessor (with the same parent) of the node designated by Position in Container. 205.2/5

```
procedure Next_Sibling (Position : in out Cursor)                                 206/5
   with Nonblocking, Global => in all, Use_Formal => null;
```

Equivalent to Position := Next_Sibling (Position); 207/3

207.1/5
```
procedure Next_Sibling (Container : in     Tree;
                        Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position));
```

207.2/5 Equivalent to Position := Next_Sibling (Container, Position);

208/5
```
procedure Previous_Sibling (Position : in out Cursor)
   with Nonblocking, Global => in all, Use_Formal => null;
```

209/3 Equivalent to Position := Previous_Sibling (Position);

209.1/5
```
procedure Previous_Sibling (Container : in     Tree;
                            Position  : in out Cursor)
   with Nonblocking, Global => null, Use_Formal => null,
        Pre  => Meaningful_For (Container, Position)
                   or else raise Program_Error,
        Post => (if Position /= No_Element
                 then Has_Element (Container, Position);
```

209.2/5 Equivalent to Position := Previous_Sibling (Container, Position);

210/5
```
procedure Iterate_Children
     (Parent  : in Cursor;
      Process : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre    => Parent /= No_Element or else raise Constraint_Error,
        Global => in all, Use_Formal => null;
```

211/5 *This paragraph was deleted.*

212/3 Iterate_Children calls Process.**all** with a cursor that designates each child node of Parent, starting with the first child node and moving the cursor as per the Next_Sibling function.

213/3 Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

213.1/5
```
procedure Iterate_Children
     (Container : in Tree;
      Parent    : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Parent /= No_Element
                   or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                   or else raise Program_Error);
```

213.2/5 Iterate_Children calls Process.**all** with a cursor that designates each child node of Container and Parent, starting with the first child node and moving the cursor as per the Next_Sibling function.

213.3/5 Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated.

214/5
```
procedure Reverse_Iterate_Children
     (Parent  : in Cursor;
      Process : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => Parent /= No_Element or else raise Constraint_Error,
        Global => in all, Use_Formal => null;
```

215/5 *This paragraph was deleted.*

Reverse_Iterate_Children calls Process.**all** with a cursor that designates each child node of Parent, starting with the last child node and moving the cursor as per the Previous_Sibling function. 216/3

Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 217/3

```
procedure Reverse_Iterate_Children                                                    217.1/5
     (Container : in Tree;
      Parent    : in Cursor;
      Process   : not null access procedure (Position : in Cursor))
   with Allows_Exit,
        Pre  => (Parent /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                    or else raise Program_Error);
```

Reverse_Iterate_Children calls Process.**all** with a cursor that designates each child node of Container and Parent, starting with the last child node and moving the cursor as per the Previous_Sibling function. 217.2/5

Tampering with the cursors of the tree containing Parent is prohibited during the execution of a call on Process.**all**. Any exception raised by Process.**all** is propagated. 217.3/5

```
function Iterate_Children (Container : in Tree; Parent : in Cursor)                  218/5
   return Tree_Iterator_Interfaces.Parallel_Reversible_Iterator'Class
   with Pre  => (Parent /= No_Element
                    or else raise Constraint_Error) and then
                (Meaningful_For (Container, Parent)
                    or else raise Program_Error),
        Post => Tampering_With_Cursors_Prohibited (Container);
```

Iterate_Children returns an iterator object (see 5.5.1) that will generate a value for a loop parameter (see 5.5.2) designating each child node of Parent. When used as a forward iterator, the nodes are designated starting with the first child node and moving the cursor as per the function Next_Sibling; when used as a reverse iterator, the nodes are designated starting with the last child node and moving the cursor as per the function Previous_Sibling; when used as a parallel iterator, processing all child nodes concurrently. Tampering with the cursors of Container is prohibited while the iterator object exists (in particular, in the sequence_of_statements of the loop_statement whose iterator_specification denotes this object). The iterator object needs finalization. 219/5

The nested package Multiway_Trees.Stable provides a type Stable.Tree that represents a *stable* tree, which is one that cannot grow and shrink. Such a tree can be created by calling the Copy function, or by establishing a *stabilized view* of an ordinary tree. 219.1/5

The subprograms of package Containers.Multiway_Trees that have a parameter or result of type tree are included in the nested package Stable with the same specification, except that the following are omitted: 219.2/5

Tampering_With_Cursors_Prohibited, Tampering_With_Elements_Prohibited, Assign, Move, Clear, Delete_Leaf, Insert_Child, Delete_Children, Delete_Subtree, Copy_Subtree, Copy_Local_Subtree, Splice_Subtree, and Splice_Children 219.3/5

The operations of this package are equivalent to those for ordinary trees, except that the calls to Tampering_With_Cursors_Prohibited and Tampering_With_Elements_Prohibited that occur in preconditions are replaced by False, and any that occur in postconditions are replaced by True. 219.4/5

219.5/5 If a stable tree is declared with the Base discriminant designating a pre-existing ordinary tree, the stable tree represents a stabilized view of the underlying ordinary tree, and any operation on the stable tree is reflected on the underlying ordinary tree. While a stabilized view exists, any operation that tampers with elements performed on the underlying tree is prohibited. The finalization of a stable tree that provides such a view removes this restriction on the underlying ordinary tree (though some other restriction can exist due to other concurrent iterations or stabilized views).

219.6/5 If a stable tree is declared without specifying Base, the object is necessarily initialized. The initializing expression of the stable tree, typically a call on Copy, determines the Node_Count of the tree. The Node_Count of a stable tree never changes after initialization.

Bounded (Run-Time) Errors

220/3 It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with elements of any Tree parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the Tree either prior to, or subsequent to, some or all of the modifications to the Tree.

221/3 It is a bounded error to call any subprogram declared in the visible part of Containers.Multiway_Trees when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

Erroneous Execution

222/3 A Cursor value is *invalid* if any of the following have occurred since it was created:

223/3 • The tree that contains the element it designates has been finalized;

224/3 • The tree that contains the element it designates has been used as the Source or Target of a call to Move;

225/3 • The tree that contains the element it designates has been used as the Target of a call to Assign or the target of an assignment_statement;

226/3 • The element it designates has been removed from the tree that previously contained the element.

227/3 The result of "=" or Has_Element is unspecified if it is called with an invalid cursor parameter. Execution is erroneous if any other subprogram declared in Containers.Multiway_Trees is called with an invalid cursor parameter.

228/3 Execution is erroneous if the tree associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

Implementation Requirements

229/3 No storage associated with a multiway tree object shall be lost upon assignment or scope exit.

230/3 The execution of an assignment_statement for a tree shall have the effect of copying the elements from the source tree object to the target tree object and changing the node count of the target object to that of the source object.

Implementation Advice

231/3 Containers.Multiway_Trees should be implemented similarly to a multiway tree. In particular, if N is the overall number of nodes for a particular tree, then the worst-case time complexity of Element, Parent, First_Child, Last_Child, Next_Sibling, Previous_Sibling, Insert_Child with Count=1, and Delete should be $O(\log N)$.

Move should not copy elements, and should minimize copying of internal data structures. 232/3

If an exception is propagated from a tree operation, no storage should be lost, nor any elements removed from a tree unless specified by the operation. 233/3

A.18.11 The Generic Package Containers.Indefinite_Vectors

The language-defined generic package Containers.Indefinite_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see A.18.2), with the difference that the generic formal Element_Type is indefinite. 1/2

Static Semantics

The declaration of the generic library package Containers.Indefinite_Vectors has the same contents and semantics as Containers.Vectors except: 2/3

- The generic formal Element_Type is indefinite. 3/2
- The procedures with the profiles: 4/2

```
procedure Insert (Container : in out Vector;
                  Before    : in     Extended_Index;
                  Count     : in     Count_Type := 1);
```
5/2

```
procedure Insert (Container : in out Vector;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```
6/2

  are omitted. 7/2
- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained. 8/2
- The operations "&", Append, Insert, Prepend, Replace_Element, and To_Vector that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 9/4
- The description of Tampering_With_Elements_Prohibited is replaced by: 10/5

  Returns True if tampering with elements is prohibited for Container, and False otherwise. 11/5
- Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference. 12/5
- The operations Replace_Element, Reverse_Elements, and Swap, and the nested generic unit Generic_Sorting are omitted from the nested package Stable. 13/5

A.18.12 The Generic Package Containers.Indefinite_Doubly_Linked_Lists

The language-defined generic package Containers.Indefinite_Doubly_Linked_Lists provides private types List and Cursor, and a set of operations for each type. It provides the same operations as the package Containers.Doubly_Linked_Lists (see A.18.3), with the difference that the generic formal Element_Type is indefinite. 1/2

Static Semantics

The declaration of the generic library package Containers.Indefinite_Doubly_Linked_Lists has the same contents and semantics as Containers.Doubly_Linked_Lists except: 2/3

- The generic formal Element_Type is indefinite. 3/2

4/2 • The procedure with the profile:

5/2
```
procedure Insert (Container : in out List;
                  Before    : in     Cursor;
                  Position  :    out Cursor;
                  Count     : in     Count_Type := 1);
```

6/2 is omitted.

7/2 • The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

8/4 • The operations Append, Insert, Prepend, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

9/5 • The description of Tampering_With_Elements_Prohibited is replaced by:

10/5 Returns True if tampering with elements is prohibited for Container, and False otherwise.

11/5 • Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference.

12/5 • The operations Replace_Element and Swap are omitted from the nested package Stable.

A.18.13 The Generic Package Containers.Indefinite_Hashed_Maps

1/2 The language-defined generic package Containers.Indefinite_Hashed_Maps provides a map with the same operations as the package Containers.Hashed_Maps (see A.18.5), with the difference that the generic formal types Key_Type and Element_Type are indefinite.

Static Semantics

2/3 The declaration of the generic library package Containers.Indefinite_Hashed_Maps has the same contents and semantics as Containers.Hashed_Maps except:

3/2 • The generic formal Key_Type is indefinite.

4/2 • The generic formal Element_Type is indefinite.

5/2 • The procedure with the profile:

6/2
```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

7/2 is omitted.

8/2 • The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

9/4 • The operations Include, Insert, Replace, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

10/5 • The description of Tampering_With_Elements_Prohibited is replaced by:

11/5 Returns True if tampering with elements is prohibited for Container, and False otherwise.

12/5 • Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference.

13/5 • The operations Replace and Replace_Element are omitted from the nested package Stable.

A.18.14 The Generic Package Containers.Indefinite_Ordered_Maps

The language-defined generic package Containers.Indefinite_Ordered_Maps provides a map with the same operations as the package Containers.Ordered_Maps (see A.18.6), with the difference that the generic formal types Key_Type and Element_Type are indefinite. 1/2

Static Semantics

The declaration of the generic library package Containers.Indefinite_Ordered_Maps has the same contents and semantics as Containers.Ordered_Maps except: 2/3

- The generic formal Key_Type is indefinite. 3/2
- The generic formal Element_Type is indefinite. 4/2
- The procedure with the profile: 5/2

```
procedure Insert (Container : in out Map;
                  Key       : in     Key_Type;
                  Position  :    out Cursor;
                  Inserted  :    out Boolean);
```

6/2

  is omitted. 7/2

- The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained. 8/2
- The operations Include, Insert, Replace, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 9/4
- The description of Tampering_With_Elements_Prohibited is replaced by: 10/5

  Returns True if tampering with elements is prohibited for Container, and False otherwise. 11/5

- Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference. 12/5
- The operations Replace and Replace_Element are omitted from the nested package Stable. 13/5

A.18.15 The Generic Package Containers.Indefinite_Hashed_Sets

The language-defined generic package Containers.Indefinite_Hashed_Sets provides a set with the same operations as the package Containers.Hashed_Sets (see A.18.8), with the difference that the generic formal type Element_Type is indefinite. 1/2

Static Semantics

The declaration of the generic library package Containers.Indefinite_Hashed_Sets has the same contents and semantics as Containers.Hashed_Sets except: 2/3

- The generic formal Element_Type is indefinite. 3/2
- The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained. 4/2
- The operations Include, Insert, Replace, Replace_Element, and To_Set that have a formal parameter of type Element_Type perform indefinite insertion (see A.18). 5/4

A.18.16 The Generic Package Containers.Indefinite_Ordered_Sets

1/2 The language-defined generic package Containers.Indefinite_Ordered_Sets provides a set with the same operations as the package Containers.Ordered_Sets (see A.18.9), with the difference that the generic formal type Element_Type is indefinite.

Static Semantics

2/3 The declaration of the generic library package Containers.Indefinite_Ordered_Sets has the same contents and semantics as Containers.Ordered_Sets except:

3/2 • The generic formal Element_Type is indefinite.

4/2 • The actual Element parameter of access subprogram Process of Update_Element_Preserving_Key may be constrained even if Element_Type is unconstrained.

5/4 • The operations Include, Insert, Replace, Replace_Element, and To_Set that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

A.18.17 The Generic Package Containers.Indefinite_Multiway_Trees

1/3 The language-defined generic package Containers.Indefinite_Multiway_Trees provides a multiway tree with the same operations as the package Containers.Multiway_Trees (see A.18.10), with the difference that the generic formal Element_Type is indefinite.

Static Semantics

2/3 The declaration of the generic library package Containers.Indefinite_Multiway_Trees has the same contents and semantics as Containers.Multiway_Trees except:

3/3 • The generic formal Element_Type is indefinite.

4/3 • The procedure with the profile:

5/3
```
procedure Insert_Child (Container : in out Tree;
                        Parent    : in     Cursor;
                        Before    : in     Cursor;
                        Position  :    out Cursor;
                        Count     : in     Count_Type := 1);
```

6/3 is omitted.

7/3 • The actual Element parameter of access subprogram Process of Update_Element may be constrained even if Element_Type is unconstrained.

8/4 • The operations Append_Child, Insert_Child, Prepend_Child, and Replace_Element that have a formal parameter of type Element_Type perform indefinite insertion (see A.18).

9/5 • The description of Tampering_With_Elements_Prohibited is replaced by:

10/5 Returns True if tampering with elements is prohibited for Container, and False otherwise.

11/5 • Tampering_With_Cursors_Prohibited is replaced by Tampering_With_Elements_Prohibited in the postcondition for the operations Reference and Constant_Reference.

12/5 • The operations Replace_Element and Swap are omitted from the nested package Stable.

A.18.18 The Generic Package Containers.Indefinite_Holders

The language-defined generic package Containers.Indefinite_Holders provides a private type Holder and a set of operations for that type. A holder container holds a single element of an indefinite type. 1/3

A holder container allows the declaration of an object that can be used like an uninitialized variable or component of an indefinite type. 2/3

A holder container may be *empty*. An empty holder does not contain an element. 3/3

Static Semantics

The generic library package Containers.Indefinite_Holders has the following declaration: 4/3

```
generic                                                                          5/5
   type Element_Type (<>) is private;
   with function "=" (Left, Right : Element_Type) return Boolean is <>;
package Ada.Containers.Indefinite_Holders
   with Preelaborate, Remote_Types,
        Nonblocking, Global => in out synchronized is

   type Holder is tagged private                                                 6/5
      with Stable_Properties => (Is_Empty,
                                 Tampering_With_The_Element_Prohibited),
           Default_Initial_Condition => Is_Empty (Holder),
           Preelaborable_Initialization;

   Empty_Holder : constant Holder;                                               7/3

   function Equal_Element (Left, Right : Element_Type) return Boolean            7.1/5
      renames "=";

   function "=" (Left, Right : Holder) return Boolean;                           8/3

   function Tampering_With_The_Element_Prohibited                                8.1/5
     (Container : Holder) return Boolean
      with Nonblocking, Global => null, Use_Formal => null;

   function Empty return Holder                                                  8.2/5
      is (Empty_Holder)
      with Post =>
            not Tampering_With_The_Element_Prohibited (Empty'Result)
            and then Is_Empty (Empty'Result);

   function To_Holder (New_Item : Element_Type) return Holder                    9/5
      with Post => not Is_Empty (To_Holder'Result);

   function Is_Empty (Container : Holder) return Boolean                         10/5
      with Global => null, Use_Formal => null;

   procedure Clear (Container : in out Holder)                                   11/5
      with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                   or else raise Program_Error,
           Post => Is_Empty (Container);

   function Element (Container : Holder) return Element_Type                     12/5
      with Pre => not Is_Empty (Container) or else raise Constraint_Error,
           Global => null, Use_Formal => Element_Type;

   procedure Replace_Element (Container : in out Holder;                         13/5
                              New_Item  : in     Element_Type)
      with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                   or else raise Program_Error,
           Post => not Is_Empty (Container);

   procedure Query_Element                                                       14/5
     (Container : in Holder;
      Process   : not null access procedure (Element : in Element_Type))
      with Pre => not Is_Empty (Container) or else raise Constraint_Error;
```

```
15/5        procedure Update_Element
               (Container : in out Holder;
                Process   : not null access procedure (Element : in out Element_Type))
                with Pre => not Is_Empty (Container) or else raise Constraint_Error;

16/6        type Constant_Reference_Type
                  (Element : not null access constant Element_Type) is
                    limited private
               with Implicit_Dereference => Element,
                    Nonblocking, Global => in out synchronized,
                    Default_Initial_Condition => (raise Program_Error);

17/6        type Reference_Type
                  (Element : not null access Element_Type) is limited private
               with Implicit_Dereference => Element,
                    Nonblocking, Global => in out synchronized,
                    Default_Initial_Condition => (raise Program_Error);

18/5        function Constant_Reference (Container : aliased in Holder)
               return Constant_Reference_Type
               with Pre  => not Is_Empty (Container)
                               or else raise Constraint_Error,
                    Post => Tampering_With_The_Element_Prohibited (Container),
                    Nonblocking, Global => null, Use_Formal => null;

19/5        function Reference (Container : aliased in out Holder)
               return Reference_Type
               with Pre  => not Is_Empty (Container)
                               or else raise Constraint_Error,
                    Post => Tampering_With_The_Element_Prohibited (Container),
                    Nonblocking, Global => null, Use_Formal => null;

20/5        procedure Assign (Target : in out Holder; Source : in Holder)
               with Post => (Is_Empty (Source) = Is_Empty (Target));

21/5        function Copy (Source : Holder) return Holder
               with Post => (Is_Empty (Source) = Is_Empty (Copy'Result));

22/5        procedure Move (Target : in out Holder; Source : in out Holder)
               with Pre  => (not Tampering_With_The_Element_Prohibited (Target)
                               or else raise Program_Error) and then
                            (not Tampering_With_The_Element_Prohibited (Source)
                               or else raise Program_Error),
                    Post => (if not Target'Has_Same_Storage (Source) then
                               Is_Empty (Source) and then (not Is_Empty (Target)));

22.1/5      procedure Swap (Left, Right : in out Holder)
               with Pre  => (not Tampering_With_The_Element_Prohibited (Left)
                               or else raise Program_Error) and then
                            (not Tampering_With_The_Element_Prohibited (Right)
                               or else raise Program_Error),
                    Post => Is_Empty (Left) = Is_Empty (Right)'Old and then
                            Is_Empty (Right) = Is_Empty (Left)'Old;

23/3    private

24/3        ... -- not specified by the language

25/3    end Ada.Containers.Indefinite_Holders;
```

26/3 The actual function for the generic formal function "=" on Element_Type values is expected to define a reflexive and symmetric relationship and return the same result value each time it is called with a particular pair of values. If it behaves in some other manner, the function "=" on holder values returns an unspecified value. The exact arguments and number of calls of this generic formal function by the function "=" on holder values are unspecified.

27/3 The type Holder is used to represent holder containers. The type Holder needs finalization (see 7.6).

28/3 Empty_Holder represents an empty holder object. If an object of type Holder is not otherwise initialized, it is initialized to the same value as Empty_Holder.

Some operations check for "tampering with the element" of a container because they depend on the element of the container not being replaced. When tampering with the element is *prohibited* for a particular holder object *H*, Program_Error is propagated by the finalization of *H*, as well as by a call that passes *H* to certain of the operations of this package, as indicated by the precondition of such an operation. 29/5

Paragraphs 30 through 35 are removed as preconditions now describe these rules.

```
function "=" (Left, Right : Holder) return Boolean;
```
36/3

If Left and Right denote the same holder object, then the function returns True. Otherwise, it compares the element contained in Left to the element contained in Right using the generic formal equality operator, returning the result of that operation. Any exception raised during the evaluation of element equality is propagated. 37/3

```
function Tampering_With_The_Element_Prohibited
  (Container : Holder) return Boolean
   with Nonblocking, Global => null, Use_Formal => null;
```
37.1/5

Returns True if tampering with the element is currently prohibited for Container, and returns False otherwise. 37.2/5

```
function To_Holder (New_Item : Element_Type) return Holder
   with Post => not Is_Empty (To_Holder'Result);
```
38/5

Returns a nonempty holder containing an element initialized to New_Item. To_Holder performs indefinite insertion (see A.18). 39/4

```
function Is_Empty (Container : Holder) return Boolean
   with Global => null, Use_Formal => null;
```
40/5

Returns True if Container is empty, and False if it contains an element. 41/3

```
procedure Clear (Container : in out Holder)
   with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                 or else raise Program_Error,
        Post => Is_Empty (Container);
```
42/5

Removes the element from Container. 43/5

```
function Element (Container : Holder) return Element_Type
   with Pre => not Is_Empty (Container) or else raise Constraint_Error,
        Global => null, Use_Formal => Element_Type;
```
44/5

Returns the element stored in Container. 45/5

```
procedure Replace_Element (Container : in out Holder;
                           New_Item  : in     Element_Type)
   with Pre  => not Tampering_With_The_Element_Prohibited (Container)
                 or else raise Program_Error,
        Post => not Is_Empty (Container);
```
46/5

Replace_Element assigns the value New_Item into Container, replacing any preexisting content of Container; Replace_Element performs indefinite insertion (see A.18). 47/5

```
procedure Query_Element
  (Container : in Holder;
   Process   : not null access procedure (Element : in Element_Type))
   with Pre => not Is_Empty (Container) or else raise Constraint_Error,
        Global => null, Use_Formal => null;
```
48/5

Query_Element calls Process.**all** with the contained element as the argument. Tampering with the element of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated. 49/5

50/5
```
procedure Update_Element
  (Container : in out Holder;
   Process   : not null access procedure (Element : in out Element_Type))
   with Pre => not Is_Empty (Container) or else raise Constraint_Error;
```

51/5 Update_Element calls Process.**all** with the contained element as the argument. Tampering with the element of Container is prohibited during the execution of the call on Process.**all**. Any exception raised by Process.**all** is propagated.

52/6
```
type Constant_Reference_Type
      (Element : not null access constant Element_Type) is
         limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

53/6
```
type Reference_Type
      (Element : not null access Element_Type) is limited private
   with Implicit_Dereference => Element,
        Nonblocking, Global => in out synchronized,
        Default_Initial_Condition => (raise Program_Error);
```

54/3 The types Constant_Reference_Type and Reference_Type need finalization.

55/5 *This paragraph was deleted.*

56/5
```
function Constant_Reference (Container : aliased in Holder)
   return Constant_Reference_Type
   with Pre  => not Is_Empty (Container) or else raise Constraint_Error,
        Post => Tampering_With_The_Element_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

57/3 This function (combined with the Implicit_Dereference aspect) provides a convenient way to gain read access to the contained element of a holder container.

58/5 Constant_Reference returns an object whose discriminant is an access value that designates the contained element. Tampering with the element of Container is prohibited while the object returned by Constant_Reference exists and has not been finalized.

59/5
```
function Reference (Container : aliased in out Holder)
   return Reference_Type
   with Pre  => not Is_Empty (Container) or else raise Constraint_Error,
        Post => Tampering_With_The_Element_Prohibited (Container),
        Nonblocking, Global => null, Use_Formal => null;
```

60/3 This function (combined with the Implicit_Dereference aspects) provides a convenient way to gain read and write access to the contained element of a holder container.

61/5 Reference returns an object whose discriminant is an access value that designates the contained element. Tampering with the element of Container is prohibited while the object returned by Reference exists and has not been finalized.

62/5
```
procedure Assign (Target : in out Holder; Source : in Holder)
   with Post => (Is_Empty (Source) = Is_Empty (Target));
```

63/3 If Target denotes the same object as Source, the operation has no effect. If Source is empty, Clear (Target) is called. Otherwise, Replace_Element (Target, Element (Source)) is called.

64/5
```
function Copy (Source : Holder) return Holder
   with Post => (Is_Empty (Source) = Is_Empty (Copy'Result));
```

65/3 If Source is empty, returns an empty holder container; otherwise, returns To_Holder (Element (Source)).

```
procedure Move (Target : in out Holder; Source : in out Holder)
   with Pre  => (not Tampering_With_The_Element_Prohibited (Target)
                    or else raise Program_Error) and then
                 (not Tampering_With_The_Element_Prohibited (Source)
                    or else raise Program_Error),
        Post => (if not Target'Has_Same_Storage (Source) then
                    Is_Empty (Source) and then (not Is_Empty (Target)));
```

66/5

If Target denotes the same object as Source, then the operation has no effect. Otherwise, the element contained by Source (if any) is removed from Source and inserted into Target, replacing any preexisting content.

67/5

```
procedure Swap (Left, Right : in out Holder)
   with Pre  => (not Tampering_With_The_Element_Prohibited (Left)
                    or else raise Program_Error) and then
                 (not Tampering_With_The_Element_Prohibited (Right)
                    or else raise Program_Error),
        Post => Is_Empty (Left) = Is_Empty (Right)'Old and then
                Is_Empty (Right) = Is_Empty (Left)'Old;
```

67.1/5

If Left denotes the same object as Right, then the operation has no effect. Otherwise, operation exchanges the elements (if any) contained by Left and Right.

67.2/5

Bounded (Run-Time) Errors

It is a bounded error for the actual function associated with a generic formal subprogram, when called as part of an operation of this package, to tamper with the element of any Holder parameter of the operation. Either Program_Error is raised, or the operation works as defined on the value of the Holder either prior to, or subsequent to, some or all of the modifications to the Holder.

68/3

It is a bounded error to call any subprogram declared in the visible part of Containers.Indefinite_Holders when the associated container has been finalized. If the operation takes Container as an **in out** parameter, then it raises Constraint_Error or Program_Error. Otherwise, the operation either proceeds as it would for an empty container, or it raises Constraint_Error or Program_Error.

69/3

Erroneous Execution

Execution is erroneous if the holder container associated with the result of a call to Reference or Constant_Reference is finalized before the result object returned by the call to Reference or Constant_Reference is finalized.

70/3

Implementation Requirements

No storage associated with a holder object shall be lost upon assignment or scope exit.

71/3

The execution of an assignment_statement for a holder container shall have the effect of copying the element (if any) from the source holder object to the target holder object.

72/3

Implementation Advice

Move and Swap should not copy any elements, and should minimize copying of internal data structures.

73/5

If an exception is propagated from a holder operation, no storage should be lost, nor should the element be removed from a holder container unless specified by the operation.

74/3

A.18.19 The Generic Package Containers.Bounded_Vectors

1/3 The language-defined generic package Containers.Bounded_Vectors provides a private type Vector and a set of operations. It provides the same operations as the package Containers.Vectors (see A.18.2), with the difference that the maximum storage is bounded.

Static Semantics

2/3 The declaration of the generic library package Containers.Bounded_Vectors has the same contents and semantics as Containers.Vectors except:

3/5 • The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

4/3 • The type Vector is declared with a discriminant that specifies the capacity:

5/5
```
type Vector (Capacity : Count_Type) is tagged private...
```

5.1/5 • The aspect_definition for Preelaborable_Initialization for type Vector is changed to:

5.2/5
```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```

6/3 • The type Vector needs finalization if and only if type Element_Type needs finalization.

6.1/5 • Capacity is omitted from the Stable_Properties of type Vector.

6.2/5 • In function Empty, the postcondition is altered to:

6.3/5
```
Post =>
   Empty'Result.Capacity = Capacity and then
   not Tampering_With_Elements_Prohibited (Empty'Result) and then
   not Tampering_With_Cursors_Prohibited (Empty'Result) and then
   Length (Empty'Result) = 0;
```

7/5 • In function Copy, the postcondition is altered to:

7.1/5
```
Post => Length (Copy'Result) = Length (Source) and then
        (if Capacity > Length (Source) then
           Copy'Result.Capacity = Capacity
         else Copy'Result.Capacity >= Length (Source));
```

8/3 • The description of Reserve_Capacity is replaced with:

8.1/5
```
procedure Reserve_Capacity (Container : in out Vector;
                            Capacity  : in     Count_Type)
   with Pre => Capacity <= Container.Capacity
                  or else raise Capacity_Error;
```

9/5 This operation has no effect, other than checking the precondition.

9.1/6 • The portion of the postcondition checking the capacity is omitted from subprograms Set_Length, Assign, Insert, Insert_Space, Slice, Replace_Slice, Prepend, Append, and Delete.

9.2/6 • For procedures Insert, Insert_Space, Replace_Slice, Prepend, and Append, the part of the precondition reading:

9.3/5
```
(<some length> <= Maximum_Length - <some other length>
   or else raise Constraint_Error)
```

9.4/5 is replaced by:

9.5/5
```
(<some length> <= Maximum_Length - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```

Bounded (Run-Time) Errors

It is a bounded error to assign from a bounded vector object while tampering with elements or cursors of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements or cursors, or execution proceeds normally. 10/3

Erroneous Execution

When a bounded vector object *V* is finalized, if tampering with cursors is prohibited for *V* other than due to an assignment from another vector, then execution is erroneous. 11/3

Implementation Requirements

For each instance of Containers.Vectors and each instance of Containers.Bounded_Vectors, if the two instances meet the following conditions, then the output generated by the Vector'Output or Vector'Write subprograms of either instance shall be readable by the Vector'Input or Vector'Read of the other instance, respectively: 12/3

- the Element_Type parameters of the two instances are statically matching subtypes of the same type; and 13/3
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters); and 14/3
- the preceding two conditions also hold for the Index_Type parameters of the instances. 15/3

Implementation Advice

Bounded vector objects should be implemented without implicit pointers or dynamic allocation. 16/3

The implementation advice for procedure Move to minimize copying does not apply. 17/3

A.18.20 The Generic Package Containers.Bounded_Doubly_Linked_Lists

The language-defined generic package Containers.Bounded_Doubly_Linked_Lists provides a private type List and a set of operations. It provides the same operations as the package Containers.Doubly_Linked_Lists (see A.18.3), with the difference that the maximum storage is bounded. 1/3

Static Semantics

The declaration of the generic library package Containers.Bounded_Doubly_Linked_Lists has the same contents and semantics as Containers.Doubly_Linked_Lists except: 2/3

- The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted. 3/5
- The type List is declared with a discriminant that specifies the capacity (maximum number of elements) as follows: 4/3

```
type List (Capacity : Count_Type) is tagged private...
```
5/5

- The aspect_definition for Preelaborable_Initialization for type List is changed to: 5.1/5

```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```
5.2/5

- The type List needs finalization if and only if type Element_Type needs finalization. 6/3
- The function Empty is replaced by: 6.1/5

6.2/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return List
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Elements_Prohibited (Empty'Result) and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Length (Empty'Result) = 0;
```

7/5 • For procedures Insert, Prepend, Append, Merge, and the three-parameter Splice whose parameter Source has type List, the part of the precondition reading:

7.1/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```

7.2/5 is replaced by:

7.3/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```

8/5 • In procedure Assign, the precondition is altered to:

8.1/5
```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
          or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
          or else raise Capacity_Error),
```

9/3 • The function Copy is replaced with:

10/5
```
function Copy (Source : List; Capacity : Count_Type := 0)
   return List
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```

11/5 Returns a list whose elements have the same values as the elements of Source.

12/5 • *This paragraph was deleted.*

13/5 • In the four-parameter procedure Splice, the precondition is altered to:

13.1/5
```
Pre  => (not Tampering_With_Cursors_Prohibited (Target)
            or else raise Program_Error) and then
         (not Tampering_With_Cursors_Prohibited (Source)
            or else raise Program_Error) and then
         (Position /= No_Element
            or else raise Constraint_Error) and then
         (Has_Element (Source, Position)
            or else raise Program_Error) and then
         (Before = No_Element or else Has_Element (Target, Before)
            or else raise Program_Error) and then
         (Target'Has_Same_Storage (Source) or else
          Length (Target) /= Count_Type'Last
            or else raise Constraint_Error) and then
         (Target'Has_Same_Storage (Source) or else
          Length (Target) /= Target.Capacity
            or else raise Capacity_Error),
```

Bounded (Run-Time) Errors

14/3 It is a bounded error to assign from a bounded list object while tampering with elements or cursors of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements or cursors, or execution proceeds normally.

Erroneous Execution

When a bounded list object *L* is finalized, if tampering with cursors is prohibited for *L* other than due to an assignment from another list, then execution is erroneous. 15/3

Implementation Requirements

For each instance of Containers.Doubly_Linked_Lists and each instance of Containers.Bounded_Doubly_Linked_Lists, if the two instances meet the following conditions, then the output generated by the List'Output or List'Write subprograms of either instance shall be readable by the List'Input or List'Read of the other instance, respectively: 16/3

- the Element_Type parameters of the two instances are statically matching subtypes of the same type; and 17/3
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters). 18/3

Implementation Advice

Bounded list objects should be implemented without implicit pointers or dynamic allocation. 19/3

The implementation advice for procedure Move to minimize copying does not apply. 20/3

A.18.21 The Generic Package Containers.Bounded_Hashed_Maps

The language-defined generic package Containers.Bounded_Hashed_Maps provides a private type Map and a set of operations. It provides the same operations as the package Containers.Hashed_Maps (see A.18.5), with the difference that the maximum storage is bounded. 1/3

Static Semantics

The declaration of the generic library package Containers.Bounded_Hashed_Maps has the same contents and semantics as Containers.Hashed_Maps except: 2/3

- The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted. 3/5
- The type Map is declared with discriminants that specify both the capacity (number of elements) and modulus (number of distinct hash values) of the hash table as follows: 4/3

```
type Map (Capacity : Count_Type;
          Modulus  : Hash_Type) is tagged private...
```
5/5

- The aspect_definition for Preelaborable_Initialization for type Map is changed to: 5.1/5

```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
      and
   Key_Type'Preelaborable_Initialization
```
5.2/5

- The type Map needs finalization if and only if type Key_Type or type Element_Type needs finalization. 6/3
- In function Empty, the postcondition is altered to: 6.1/5

```
Post =>
   Empty'Result.Capacity = Capacity and then
   Empty'Result.Modulus = Default_Modulus (Capacity) and then
   not Tampering_With_Elements_Prohibited (Empty'Result) and then
   not Tampering_With_Cursors_Prohibited (Empty'Result) and then
   Length (Empty'Result) = 0;
```
6.2/5

- The description of Reserve_Capacity is replaced with: 7/3

7.1/5
```
procedure Reserve_Capacity (Container : in out Map;
                            Capacity  : in     Count_Type)
   with Pre => Capacity <= Container.Capacity
                  or else raise Capacity_Error;
```

8/5 This operation has no effect, other than checking the precondition.

9/3 • An additional operation is added immediately following Reserve_Capacity:

10/3
```
function Default_Modulus (Capacity : Count_Type) return Hash_Type;
```

11/3 Default_Modulus returns an implementation-defined value for the number of distinct hash values to be used for the given capacity (maximum number of elements).

11.1/5 • For procedures Insert and Include, the part of the precondition reading:

11.2/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```

11.3/5 is replaced by:

11.4/5
```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> > Container.Capacity - <some other length>
   or else raise Capacity_Error)
```

11.5/5 • In procedure Assign, the precondition is altered to:

11.6/5
```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
           or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
           or else raise Capacity_Error),
```

12/3 • The function Copy is replaced with:

13/5
```
function Copy (Source   : Map;
               Capacity : Count_Type := 0;
               Modulus  : Hash_Type := 0) return Map
  with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                  or else raise Capacity_Error,
       Post =>
          Length (Copy'Result) = Length (Source) and then
          not Tampering_With_Elements_Prohibited (Copy'Result) and then
          not Tampering_With_Cursors_Prohibited (Copy'Result) and then
          Copy'Result.Capacity = (if Capacity = 0 then
             Length (Source) else Capacity) and then
          Copy'Result.Modulus = (if Modulus = 0 then
             Default_Modulus (Capacity) else Modulus);
```

14/5 Returns a map with key/element pairs initialized from the values in Source.

Bounded (Run-Time) Errors

15/3 It is a bounded error to assign from a bounded map object while tampering with elements or cursors of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements or cursors, or execution proceeds normally.

Erroneous Execution

16/3 When a bounded map object *M* is finalized, if tampering with cursors is prohibited for *M* other than due to an assignment from another map, then execution is erroneous.

Implementation Requirements

17/3 For each instance of Containers.Hashed_Maps and each instance of Containers.Bounded_Hashed_Maps, if the two instances meet the following conditions, then the output generated by the Map'Output or Map'Write subprograms of either instance shall be readable by the Map'Input or Map'Read of the other instance, respectively:

- the Element_Type parameters of the two instances are statically matching subtypes of the same type; and 18/3
- the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters); and 19/3
- the preceding two conditions also hold for the Key_Type parameters of the instances. 20/3

Implementation Advice

Bounded hashed map objects should be implemented without implicit pointers or dynamic allocation. 21/3

The implementation advice for procedure Move to minimize copying does not apply. 22/3

A.18.22 The Generic Package Containers.Bounded_Ordered_Maps

The language-defined generic package Containers.Bounded_Ordered_Maps provides a private type Map and a set of operations. It provides the same operations as the package Containers.Ordered_Maps (see A.18.6), with the difference that the maximum storage is bounded. 1/3

Static Semantics

The declaration of the generic library package Containers.Bounded_Ordered_Maps has the same contents and semantics as Containers.Ordered_Maps except: 2/3

- The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted. 3/5
- The type Map is declared with a discriminant that specifies the capacity (maximum number of elements) as follows: 4/3

```
type Map (Capacity : Count_Type) is tagged private...
```
5/5

- The aspect_definition for Preelaborable_Initialization for type Map is changed to: 5.1/5

```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
      and
   Key_Type'Preelaborable_Initialization
```
5.2/5

- The type Map needs finalization if and only if type Key_Type or type Element_Type needs finalization. 6/3
- The function Empty is replaced by: 6.1/5

```
function Empty (Capacity : Count_Type := implementation-defined)
   return Map
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Elements_Prohibited (Empty'Result) and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Length (Empty'Result) = 0;
```
6.2/5

- For procedures Insert and Include, the part of the precondition reading: 7/5

```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```
7.1/5

  is replaced by: 7.2/5

```
(<some length> <= Count_Type'Last - <some other length>
    or else raise Constraint_Error) and then
 (<some length> <= Container.Capacity - <some other length>
    or else raise Capacity_Error)
```
7.3/5

- In procedure Assign, the precondition is altered to: 8/5

8.1/5
```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
          or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
          or else raise Capacity_Error),
```

9/3 • The function Copy is replaced with:

10/5
```
function Copy (Source   : Map;
               Capacity : Count_Type := 0) return Map
  with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                  or else raise Capacity_Error,
       Post =>
          Length (Copy'Result) = Length (Source) and then
          not Tampering_With_Elements_Prohibited (Copy'Result) and then
          not Tampering_With_Cursors_Prohibited (Copy'Result) and then
          Copy'Result.Capacity = (if Capacity = 0 then
             Length (Source) else Capacity);
```

11/5 Returns a map with key/element pairs initialized from the values in Source.

Bounded (Run-Time) Errors

12/3 It is a bounded error to assign from a bounded map object while tampering with elements or cursors of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements or cursors, or execution proceeds normally.

Erroneous Execution

13/3 When a bounded map object *M* is finalized, if tampering with cursors is prohibited for *M* other than due to an assignment from another map, then execution is erroneous.

Implementation Requirements

14/3 For each instance of Containers.Ordered_Maps and each instance of Containers.Bounded_Ordered_Maps, if the two instances meet the following conditions, then the output generated by the Map'Output or Map'Write subprograms of either instance shall be readable by the Map'Input or Map'Read of the other instance, respectively:

15/3 • the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

16/3 • the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters); and

17/3 • the preceding two conditions also hold for the Key_Type parameters of the instances.

Implementation Advice

18/3 Bounded ordered map objects should be implemented without implicit pointers or dynamic allocation.

19/3 The implementation advice for procedure Move to minimize copying does not apply.

A.18.23 The Generic Package Containers.Bounded_Hashed_Sets

1/3 The language-defined generic package Containers.Bounded_Hashed_Sets provides a private type Set and a set of operations. It provides the same operations as the package Containers.Hashed_Sets (see A.18.8), with the difference that the maximum storage is bounded.

Static Semantics

The declaration of the generic library package Containers.Bounded_Hashed_Sets has the same contents and semantics as Containers.Hashed_Sets except: 2/3

- The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted. 3/5
- The type Set is declared with discriminants that specify both the capacity (number of elements) and modulus (number of distinct hash values) of the hash table as follows: 4/3

```
type Set (Capacity : Count_Type;
          Modulus  : Hash_Type) is tagged private...
```
5/5

- The aspect_definition for Preelaborable_Initialization for type Set is changed to: 5.1/5

```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```
5.2/5

- The type Set needs finalization if and only if type Element_Type needs finalization. 6/3
- In function Empty, the postcondition is altered to: 6.1/5

```
Post =>
   Empty'Result.Capacity = Capacity and then
   Empty'Result.Modulus = Default_Modulus (Capacity) and then
   not Tampering_With_Cursors_Prohibited (Empty'Result) and then
   Length (Empty'Result) = 0;
```
6.2/5

- The description of Reserve_Capacity is replaced with: 7/3

```
procedure Reserve_Capacity (Container : in out Set;
                            Capacity  : in     Count_Type)
   with Pre => Capacity <= Container.Capacity
      or else raise Capacity_Error;
```
7.1/5

  This operation has no effect, other than checking the precondition. 8/5

- An additional operation is added immediately following Reserve_Capacity: 9/3

```
function Default_Modulus (Capacity : Count_Type) return Hash_Type;
```
10/3

  Default_Modulus returns an implementation-defined value for the number of distinct hash values to be used for the given capacity (maximum number of elements). 11/3

- For procedures Insert and Include, the part of the precondition reading: 11.1/5

```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```
11.2/5

  is replaced by: 11.3/5

```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```
11.4/5

- In procedure Assign, the precondition is altered to: 11.5/5

```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
          or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
          or else raise Capacity_Error),
```
11.6/5

- The function Copy is replaced with: 12/3

13/5
```
function Copy (Source   : Set;
               Capacity : Count_Type := 0;
               Modulus  : Hash_Type := 0) return Map
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity) and then
           Copy'Result.Modulus = (if Modulus = 0 then
              Default_Modulus (Capacity) else Modulus);
```

14/5 Returns a set with key/element pairs initialized from the values in Source.

Bounded (Run-Time) Errors

15/3 It is a bounded error to assign from a bounded set object while tampering with elements or cursors of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements or cursors, or execution proceeds normally.

Erroneous Execution

16/3 When a bounded set object *S* is finalized, if tampering with cursors is prohibited for *S* other than due to an assignment from another set, then execution is erroneous.

Implementation Requirements

17/3 For each instance of Containers.Hashed_Sets and each instance of Containers.Bounded_Hashed_Sets, if the two instances meet the following conditions, then the output generated by the Set'Output or Set'Write subprograms of either instance shall be readable by the Set'Input or Set'Read of the other instance, respectively:

18/3 • the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

19/3 • the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters).

Implementation Advice

20/3 Bounded hashed set objects should be implemented without implicit pointers or dynamic allocation.

21/3 The implementation advice for procedure Move to minimize copying does not apply.

A.18.24 The Generic Package Containers.Bounded_Ordered_Sets

1/3 The language-defined generic package Containers.Bounded_Ordered_Sets provides a private type Set and a set of operations. It provides the same operations as the package Containers.Ordered_Sets (see A.18.9), with the difference that the maximum storage is bounded.

Static Semantics

2/3 The declaration of the generic library package Containers.Bounded_Ordered_Sets has the same contents and semantics as Containers.Ordered_Sets except:

3/5 • The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

4/3 • The type Set is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

```
type Set (Capacity : Count_Type) is tagged private...
```
5/5

- The aspect_definition for Preelaborable_Initialization for type Set is changed to: 5.1/5

```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```
5.2/5

- The type Set needs finalization if and only if type Element_Type needs finalization. 6/3

- The function Empty is replaced by: 6.1/5

```
function Empty (Capacity : Count_Type := implementation-defined)
   return Set
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Length (Empty'Result) = 0;
```
6.2/5

- For procedures Insert and Include, the part of the precondition reading: 7/5

```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error)
```
7.1/5

  is replaced by: 7.2/5

```
(<some length> <= Count_Type'Last - <some other length>
   or else raise Constraint_Error) and then
(<some length> <= Container.Capacity - <some other length>
   or else raise Capacity_Error)
```
7.3/5

- In procedure Assign, the precondition is altered to: 8/5

```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
          or else raise Program_Error) and then
       (Length (Source) <= Target.Capacity
          or else raise Capacity_Error),
```
8.1/5

- The function Copy is replaced with: 9/3

```
function Copy (Source   : Set;
               Capacity : Count_Type := 0) return Map
   with Pre  => Capacity = 0 or else Capacity >= Length (Source)
                   or else raise Capacity_Error,
        Post =>
           Length (Copy'Result) = Length (Source) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Length (Source) else Capacity);
```
10/5

  Returns a set with key/element pairs initialized from the values in Source. 11/5

Bounded (Run-Time) Errors

It is a bounded error to assign from a bounded set object while tampering with elements or cursors of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements or cursors, or execution proceeds normally. 12/3

Erroneous Execution

When a bounded set object *S* is finalized, if tampering with cursors is prohibited for *S* other than due to an assignment from another set, then execution is erroneous. 13/3

Implementation Requirements

For each instance of Containers.Ordered_Sets and each instance of Containers.Bounded_Ordered_Sets, if the two instances meet the following conditions, then the output generated by the Set'Output or Set'Write subprograms of either instance shall be readable by the Set'Input or Set'Read of the other instance, respectively: 14/3

15/3 • the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

16/3 • the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters).

Implementation Advice

17/3 Bounded ordered set objects should be implemented without implicit pointers or dynamic allocation.

18/3 The implementation advice for procedure Move to minimize copying does not apply.

A.18.25 The Generic Package Containers.Bounded_Multiway_Trees

1/3 The language-defined generic package Containers.Bounded_Multiway_Trees provides a private type Tree and a set of operations. It provides the same operations as the package Containers.Multiway_Trees (see A.18.10), with the difference that the maximum storage is bounded.

Static Semantics

2/3 The declaration of the generic library package Containers.Bounded_Multiway_Trees has the same contents and semantics as Containers.Multiway_Trees except:

3/5 • The aspect Preelaborate is replaced with aspect Pure. Aspect Global is deleted.

4/3 • The type Tree is declared with a discriminant that specifies the capacity (maximum number of elements) as follows:

5/5
```
type Tree (Capacity : Count_Type) is tagged private...
```

5.1/5 • The aspect_definition for Preelaborable_Initialization for type Tree is changed to:

5.2/5
```
Preelaborable_Initialization =>
   Element_Type'Preelaborable_Initialization
```

6/3 • The type Tree needs finalization if and only if type Element_Type needs finalization.

6.1/5 • The function Empty is replaced by:

6.2/5
```
function Empty (Capacity : Count_Type := implementation-defined)
   return Tree
   with Post =>
         Empty'Result.Capacity = Capacity and then
         not Tampering_With_Elements_Prohibited (Empty'Result) and then
         not Tampering_With_Cursors_Prohibited (Empty'Result) and then
         Node_Count (Empty'Result) = 1;
```

7/5 • For procedures Insert_Child, Prepend_Child, and Append_Child, the initial subexpression of the precondition is replaced with:

7.1/5
```
with Pre => (not Tampering_With_Cursors_Prohibited (Container)
                or else raise Program_Error) and then
             (Node_Count (Container) - 1 <= Container.Capacity - Count
                or else raise Capacity_Error)
```

8/5 • In procedure Assign, the precondition is altered to:

8.1/5
```
Pre => (not Tampering_With_Cursors_Prohibited (Target)
           or else raise Program_Error) and then
       (Node_Count (Source) - 1 <= Target.Capacity
           or else raise Capacity_Error),
```

9/3 • Function Copy is declared as follows:

```
function Copy (Source : Tree; Capacity : Count_Type := 0)                    10/4
   return Tree
   with Pre  => Capacity = 0 or else Capacity >= Node_Count (Source) - 1
                   or else raise Capacity_Error,
        Post =>
           Node_Count (Copy'Result) = Node_Count (Source) and then
           not Tampering_With_Elements_Prohibited (Copy'Result) and then
           not Tampering_With_Cursors_Prohibited (Copy'Result) and then
           Copy'Result.Capacity = (if Capacity = 0 then
              Node_Count (Source) - 1 else Capacity);
```

Returns a list whose elements have the same values as the elements of Source. 11/5

- In the four-parameter procedure Copy_Subtree, the last **or else** of the precondition is replaced by: 11.1/5

```
(not Is_Root (Source)                                                        11.2/5
   or else raise Constraint_Error) and then
(Node_Count (Target) - 1 + Subtree_Node_Count (Source) <=
   Target.Capacity
   or else raise Capacity_Error),
```

- In the five-parameter procedure Copy_Subtree, the last **or else** of the precondition is replaced by: 11.3/5

```
(not Is_Root (Source, Subtree)                                               11.4/5
   or else raise Constraint_Error) and then
(Node_Count (Target) - 1 +
      Subtree_Node_Count (Source, Subtree) <= Target.Capacity
   or else raise Capacity_Error),
```

- In Copy_Local_Subtree, the last **or else** of the precondition is replaced by: 11.5/5

```
(not Is_Root (Source, Subtree)                                               11.6/5
   or else raise Constraint_Error) and then
(Node_Count (Target) - 1 +
      Subtree_Node_Count (Target, Source) <= Target.Capacity
   or else raise Capacity_Error),
```

- In the five-parameter procedure Splice_Subtree, the penultimate **or else** of the precondition is replaced by: 12/5

```
(Has_Element (Source, Position)                                              12.1/5
   or else raise Program_Error) and then
(Target'Has_Same_Storage (Source) or else
 Node_Count (Target) - 1 +
   Subtree_Node_Count (Source, Position) <= Target.Capacity
   or else raise Capacity_Error) and then
```

- In the five-parameter procedure Splice_Children, the penultimate **elsif** of the precondition is replaced by: 13/5

```
(Before = No_Element or else                                                 13.1/5
 Parent (Target, Before) /= Target_Parent
   or else raise Constraint_Error) and then
(Target'Has_Same_Storage (Source) or else
 Node_Count (Target) - 1 +
   Child_Count (Source, Source_Parent) <= Target.Capacity
   or else raise Capacity_Error) and then
```

Bounded (Run-Time) Errors

It is a bounded error to assign from a bounded tree object while tampering with elements or cursors of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements or cursors, or execution proceeds normally. 14/3

Erroneous Execution

15/3 When a bounded tree object *T* is finalized, if tampering with cursors is prohibited for *T* other than due to an assignment from another tree, then execution is erroneous.

Implementation Requirements

16/3 For each instance of Containers.Multiway_Trees and each instance of Containers.Bounded_Multiway_Trees, if the two instances meet the following conditions, then the output generated by the Tree'Output or Tree'Write subprograms of either instance shall be readable by the Tree'Input or Tree'Read of the other instance, respectively:

17/3 - the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

18/3 - the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters).

Implementation Advice

19/3 Bounded tree objects should be implemented without implicit pointers or dynamic allocation.

20/3 The implementation advice for procedure Move to minimize copying does not apply.

A.18.26 Array Sorting

1/3 The language-defined generic procedures Containers.Generic_Array_Sort, Containers.Generic_Constrained_Array_Sort, and Containers.Generic_Sort provide sorting on arbitrary array types.

Static Semantics

2/2 The generic library procedure Containers.Generic_Array_Sort has the following declaration:

3/5
```
generic
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type range <>) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Array_Sort (Container : in out Array_Type)
   with Pure, Nonblocking, Global => null;
```

4/2 Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated.

5/3 The actual function for the generic formal function "<" of Generic_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Array_Sort is unspecified. The number of times Generic_Array_Sort calls "<" is unspecified.

The generic library procedure Containers.Generic_Constrained_Array_Sort has the following declaration: 6/2

```
generic                                                                    7/5
   type Index_Type is (<>);
   type Element_Type is private;
   type Array_Type is array (Index_Type) of Element_Type;
   with function "<" (Left, Right : Element_Type)
      return Boolean is <>;
procedure Ada.Containers.Generic_Constrained_Array_Sort
      (Container : in out Array_Type)
   with Pure, Nonblocking, Global => null;
```

Reorders the elements of Container such that the elements are sorted smallest first as determined by the generic formal "<" operator provided. Any exception raised during evaluation of "<" is propagated. 8/2

The actual function for the generic formal function "<" of Generic_Constrained_Array_Sort is expected to return the same value each time it is called with a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify Container. If the actual for "<" behaves in some other manner, the behavior of the instance of Generic_Constrained_Array_Sort is unspecified. The number of times Generic_Constrained_Array_Sort calls "<" is unspecified. 9/3

The generic library procedure Containers.Generic_Sort has the following declaration: 9.1/3

```
generic                                                                    9.2/5
   type Index_Type is (<>);
   with function Before (Left, Right : Index_Type) return Boolean;
   with procedure Swap (Left, Right : in Index_Type);
procedure Ada.Containers.Generic_Sort
      (First, Last : Index_Type'Base)
   with Pure, Nonblocking, Global => null;
```

Reorders the elements of an indexable structure, over the range First .. Last, such that the elements are sorted in the ordering determined by the generic formal function Before; Before should return True if Left is to be sorted before Right. The generic formal Before compares the elements having the given indices, and the generic formal Swap exchanges the values of the indicated elements. Any exception raised during evaluation of Before or Swap is propagated. 9.3/3

The actual function for the generic formal function Before of Generic_Sort is expected to return the same value each time it is called with index values that identify a particular pair of element values. It should define a strict weak ordering relationship (see A.18); it should not modify the elements. The actual function for the generic formal Swap should exchange the values of the indicated elements. If the actual for either Before or Swap behaves in some other manner, the behavior of Generic_Sort is unspecified. The number of times the Generic_Sort calls Before or Swap is unspecified. 9.4/3

Implementation Advice

The worst-case time complexity of a call on an instance of Containers.Generic_Array_Sort or Containers.Generic_Constrained_Array_Sort should be *O*(*N***2) or better, and the average time complexity should be better than *O*(*N***2), where *N* is the length of the Container parameter. 10/2

Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements. 11/2

The worst-case time complexity of a call on an instance of Containers.Generic_Sort should be *O*(*N***2) or better, and the average time complexity should be better than *O*(*N***2), where *N* is the difference between the Last and First parameters plus 1. 12/3

13/3 Containers.Generic_Sort should minimize calls to the generic formal Swap.

A.18.27 The Generic Package Containers.Synchronized_Queue_Interfaces

1/3 The language-defined generic package Containers.Synchronized_Queue_Interfaces provides interface type Queue, and a set of operations for that type. Interface Queue specifies a first-in, first-out queue.

Static Semantics

2/3 The generic library package Containers.Synchronized_Queue_Interfaces has the following declaration:

3/5
```
generic
   type Element_Type is private;
package Ada.Containers.Synchronized_Queue_Interfaces
   with Pure, Nonblocking, Global => null is
```

4/3
```
   type Queue is synchronized interface;
```

5/5
```
   procedure Enqueue
     (Container : in out Queue;
      New_Item  : in     Element_Type) is abstract
       with Synchronization => By_Entry,
            Nonblocking => False,
            Global'Class=> in out synchronized;
```

6/5
```
   procedure Dequeue
     (Container : in out Queue;
      Element   :    out Element_Type) is abstract
       with Synchronization => By_Entry,
            Nonblocking => False,
            Global'Class=> in out synchronized;
```

7/5
```
   function Current_Use (Container : Queue) return Count_Type is abstract
       with Nonblocking, Global'Class => null, Use_Formal => null;
   function Peak_Use (Container : Queue) return Count_Type is abstract
       with Nonblocking, Global'Class => null, Use_Formal => null,
            Post'Class => Peak_Use'Result >= Current_Use (Container);
```

8/3
```
end Ada.Containers.Synchronized_Queue_Interfaces;
```

8.1/5 The subprogram behavior descriptions given below are the semantics for the corresponding callable entities found in the language-defined generic packages that have a formal package named Queue_Interfaces.

9/5
```
procedure Enqueue
  (Container : in out Queue;
   New_Item  : in     Element_Type) is abstract
   with Synchronization => By_Entry
        Nonblocking => False,
        Global'Class=> in out synchronized;
```

10/3 A queue type that implements this interface is allowed to have a bounded *capacity*. If the queue object has a bounded capacity, and the number of existing elements equals the capacity, then Enqueue blocks until storage becomes available; otherwise, Enqueue does not block. In any case, it then copies New_Item onto the queue.

```
procedure Dequeue                                                          11/5
  (Container : in out Queue;
   Element   :    out Element_Type) is abstract
   with Synchronization => By_Entry
        Nonblocking => False,
        Global'Class=> in out synchronized;
```

If the queue is empty, then Dequeue blocks until an item becomes available. In any case, it then assigns the element at the head of the queue to Element, and removes it from the queue. 12/3

```
function Current_Use (Container : Queue) return Count_Type is abstract     13/5
   with Nonblocking, Global'Class=> null, Use_Formal => null;
```

Returns the number of elements currently in the queue. 14/3

```
function Peak_Use (Container : Queue) return Count_Type is abstract        15/5
   with Nonblocking, Global'Class=> null, Use_Formal => null,
         Post'Class => Peak_Use'Result >= Current_Use (Container);
```

Returns the maximum number of elements that have been in the queue at any one time. 16/3

NOTE Unlike other language-defined containers, there are no queues whose element types are indefinite. Elements of an indefinite type can be handled by defining the element of the queue to be a holder container (see A.18.18) of the indefinite type, or to be an explicit access type that designates the indefinite type. 17/3

A.18.28 The Generic Package Containers.Unbounded_Synchronized_Queues

Static Semantics

The language-defined generic package Containers.Unbounded_Synchronized_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue. 1/3

```
with System;                                                               2/5
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is
      new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   Default_Ceiling : System.Any_Priority := System.Priority'Last;
package Ada.Containers.Unbounded_Synchronized_Queues
   with Preelaborate,
        Nonblocking, Global => in out synchronized is

   package Implementation is                                               3/3
      ... -- not specified by the language
   end Implementation;

   protected type Queue                                                    4/3
        (Ceiling : System.Any_Priority := Default_Ceiling)
           with Priority => Ceiling is
        new Queue_Interfaces.Queue with

      overriding                                                           5/3
      entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element : out Queue_Interfaces.Element_Type);

      overriding                                                           6/5
      function Current_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
      overriding
      function Peak_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;

   private                                                                 7/3
      ... -- not specified by the language
   end Queue;
```

```
8/3     private
9/3        ... -- not specified by the language
10/3    end Ada.Containers.Unbounded_Synchronized_Queues;
```

11/3 The type Queue is used to represent task-safe queues.

12/3 The capacity for instances of type Queue is unbounded.

A.18.29 The Generic Package Containers.Bounded_Synchronized_Queues

Static Semantics

1/3 The language-defined generic package Containers.Bounded_Synchronized_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

```
2/6     with System;
        with Ada.Containers.Synchronized_Queue_Interfaces;
        generic
           with package Queue_Interfaces is
              new Ada.Containers.Synchronized_Queue_Interfaces (<>);
           Default_Capacity : Count_Type;
           Default_Ceiling  : System.Any_Priority := System.Priority'Last;
           Max_Capacity : Count_Type := Count_Type'Last;
        package Ada.Containers.Bounded_Synchronized_Queues
           with Preelaborate,
                Nonblocking, Global => in out synchronized is

3/3        package Implementation is
              ... -- not specified by the language
           end Implementation;

3.1/6      subtype Bounded_Capacity is
              Count_Type range Count_Type'First .. Max_Capacity;

4/6        protected type Queue
                (Capacity : Bounded_Capacity := Default_Capacity;
                 Ceiling  : System.Any_Priority := Default_Ceiling)
                   with Priority => Ceiling is
                new Queue_Interfaces.Queue with

5/3           overriding
              entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
              overriding
              entry Dequeue (Element : out Queue_Interfaces.Element_Type);

6/5           overriding
              function Current_Use return Count_Type
                 with Nonblocking, Global => null, Use_Formal => null;
              overriding
              function Peak_Use return Count_Type
                 with Nonblocking, Global => null, Use_Formal => null;

7/3        private
              ... -- not specified by the language
           end Queue;

8/3     private
9/3        ... -- not specified by the language
10/3    end Ada.Containers.Bounded_Synchronized_Queues;
```

11/3 The semantics are the same as for Unbounded_Synchronized_Queues, except:

12/3 • The capacity for instances of type Queue is bounded and specified by the discriminant Capacity.

Implementation Advice

Bounded queue objects should be implemented without implicit pointers or dynamic allocation. 13/3

A.18.30 The Generic Package Containers.Unbounded_Priority_Queues

Static Semantics

The language-defined generic package Containers.Unbounded_Priority_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue. 1/3

```
with System;                                                                    2/5
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is
      new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   type Queue_Priority is private;
   with function Get_Priority
     (Element : Queue_Interfaces.Element_Type) return Queue_Priority is <>;
   with function Before
     (Left, Right : Queue_Priority) return Boolean is <>;
   Default_Ceiling : System.Any_Priority := System.Priority'Last;
package Ada.Containers.Unbounded_Priority_Queues
   with Preelaborate,
        Nonblocking, Global => in out synchronized is

   package Implementation is                                                    3/3
      ... -- not specified by the language
   end Implementation;

   protected type Queue                                                         4/3
        (Ceiling : System.Any_Priority := Default_Ceiling)
           with Priority => Ceiling is
        new Queue_Interfaces.Queue with

      overriding                                                                5/3
      entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element : out Queue_Interfaces.Element_Type);

      not overriding                                                            6/3
      procedure Dequeue_Only_High_Priority
        (At_Least : in     Queue_Priority;
         Element  : in out Queue_Interfaces.Element_Type;
         Success  :    out Boolean);

      overriding                                                                7/5
      function Current_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
      overriding
      function Peak_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;

   private                                                                      8/3
      ... -- not specified by the language
   end Queue;

private                                                                         9/3

   ... -- not specified by the language                                         10/3

end Ada.Containers.Unbounded_Priority_Queues;                                   11/3
```

The type Queue is used to represent task-safe priority queues. 12/3

The capacity for instances of type Queue is unbounded. 13/3

Two elements *E1* and *E2* are equivalent if Before(Get_Priority(*E1*), Get_Priority(*E2*)) and Before(Get_Priority(*E2*), Get_Priority(*E1*)) both return False. 14/3

15/3 The actual functions for Get_Priority and Before are expected to return the same value each time they are called with the same actuals, and should not modify their actuals. Before should define a strict weak ordering relationship (see A.18). If the actual functions behave in some other manner, the behavior of Unbounded_Priority_Queues is unspecified.

16/3 Enqueue inserts an item according to the order specified by the Before function on the result of Get_Priority on the elements; Before should return True if Left is to be inserted before Right. If the queue already contains elements equivalent to New_Item, then it is inserted after the existing equivalent elements.

17/3 For a call on Dequeue_Only_High_Priority, if the head of the nonempty queue is *E*, and the function Before(At_Least, Get_Priority(*E*)) returns False, then *E* is assigned to Element and then removed from the queue, and Success is set to True; otherwise, Success is set to False and Element is unchanged.

A.18.31 The Generic Package Containers.Bounded_Priority_Queues

Static Semantics

1/3 The language-defined generic package Containers.Bounded_Priority_Queues provides type Queue, which implements the interface type Containers.Synchronized_Queue_Interfaces.Queue.

2/6
```
with System;
with Ada.Containers.Synchronized_Queue_Interfaces;
generic
   with package Queue_Interfaces is
      new Ada.Containers.Synchronized_Queue_Interfaces (<>);
   type Queue_Priority is private;
   with function Get_Priority
     (Element : Queue_Interfaces.Element_Type) return Queue_Priority is <>;
   with function Before
     (Left, Right : Queue_Priority) return Boolean is <>;
   Default_Capacity : Count_Type;
   Default_Ceiling  : System.Any_Priority := System.Priority'Last;
   Max_Capacity : Count_Type := Count_Type'Last;
   package Ada.Containers.Bounded_Priority_Queues
   with Preelaborate,
        Nonblocking, Global => in out synchronized is
```

3/3
```
   package Implementation is
      ... -- not specified by the language
   end Implementation;
```

3.1/6
```
   subtype Bounded_Capacity is
      Count_Type range Count_Type'First .. Max_Capacity;
```

4/6
```
   protected type Queue
        (Capacity : Bounded_Capacity := Default_Capacity;
         Ceiling  : System.Any_Priority := Default_Ceiling)
           with Priority => Ceiling is
      new Queue_Interfaces.Queue with
```

5/3
```
      overriding
      entry Enqueue (New_Item : in Queue_Interfaces.Element_Type);
      overriding
      entry Dequeue (Element : out Queue_Interfaces.Element_Type);
```

6/3
```
      not overriding
      procedure Dequeue_Only_High_Priority
        (At_Least : in     Queue_Priority;
         Element  : in out Queue_Interfaces.Element_Type;
         Success  :    out Boolean);
```

```
      overriding                                                              7/5
      function Current_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
      overriding
      function Peak_Use return Count_Type
         with Nonblocking, Global => null, Use_Formal => null;
   private                                                                    8/3
      ... -- not specified by the language
   end Queue;
private                                                                       9/3
   ... -- not specified by the language                                       10/3
end Ada.Containers.Bounded_Priority_Queues;                                   11/3
```

The semantics are the same as for Unbounded_Priority_Queues, except: 12/3

- The capacity for instances of type Queue is bounded and specified by the discriminant Capacity. 13/3

Implementation Advice

Bounded priority queue objects should be implemented without implicit pointers or dynamic allocation. 14/3

A.18.32 The Generic Package Containers.Bounded_Indefinite_Holders

The language-defined generic package Containers.Bounded_Indefinite_Holders provides a private type Holder and a set of operations for that type. It provides the same operations as the package Containers.Indefinite_Holders (see A.18.18), with the difference that the maximum storage is bounded. 1/5

Static Semantics

The declaration of the generic library package Containers.Bounded_Indefinite_Holders has the same contents and semantics as Containers.Indefinite_Holders except: 2/5

- The following is added to the context clause: 3/5

```
with System.Storage_Elements; use System.Storage_Elements;                    4/5
```

- An additional generic parameter follows Element_Type: 5/5

```
Max_Element_Size_in_Storage_Elements : Storage_Count;                         6/5
```

- The aspect_definition for Preelaborable_Initialization for type Holder is changed to: 7/5

```
Preelaborable_Initialization =>                                               8/5
   Element_Type'Preelaborable_Initialization
```

- Add to the precondition of To_Holder and Replace_Element: 9/5

```
and then (New_Item'Size <=                                                    10/5
              Max_Element_Size_in_Storage_Elements * System.Storage_Unit
          or else raise Program_Error)
```

Bounded (Run-Time) Errors

It is a bounded error to assign from a bounded holder object while tampering with elements of that object is prohibited. Either Program_Error is raised by the assignment, execution proceeds with the target object prohibiting tampering with elements, or execution proceeds normally. 11/5

Implementation Requirements

For each instance of Containers.Indefinite_Holders and each instance of Containers.Bounded_-Indefinite_Holders, if the two instances meet the following conditions, then the output generated by the Holder'Output or Holder'Write subprograms of either instance shall be readable by the Holder'Input or Holder'Read of the other instance, respectively: 12/5

13/5 • the Element_Type parameters of the two instances are statically matching subtypes of the same type; and

14/5 • the output generated by Element_Type'Output or Element_Type'Write is readable by Element_Type'Input or Element_Type'Read, respectively (where Element_Type denotes the type of the two actual Element_Type parameters).

Implementation Advice

15/5 Bounded holder objects should be implemented without dynamic allocation and any finalization should be trivial unless Element_Type needs finalization.

16/5 The Implementation Advice about the Move and Swap operations is deleted for bounded holders; these operations can copy elements as necessary.

A.18.33 Example of Container Use

Examples

1/3 The following example is an implementation of Dijkstra's shortest path algorithm in a directed graph with positive distances. The graph is represented by a map from nodes to sets of edges.

2/3
```
with Ada.Containers.Vectors;
with Ada.Containers.Doubly_Linked_Lists;
use Ada.Containers;
generic
   type Node is range <>;
package Shortest_Paths is
   type Distance is new Float range 0.0 .. Float'Last;
   type Edge is record
      To, From : Node;
      Length   : Distance;
   end record;
```

3/3
```
   package Node_Maps is new Vectors (Node, Node);
   -- The algorithm builds a map to indicate the node used to reach a given
   -- node in the shortest distance.
```

4/3
```
   package Adjacency_Lists is new Doubly_Linked_Lists (Edge);
   use Adjacency_Lists;
```

5/3
```
   package Graphs is new Vectors (Node, Adjacency_Lists.List);
```

6/3
```
   package Paths is new Doubly_Linked_Lists (Node);
```

7/3
```
   function Shortest_Path
     (G : Graphs.Vector; Source : Node; Target : Node) return Paths.List
      with Pre => G (Source) /= Adjacency_Lists.Empty_List;
```

8/3
```
end Shortest_Paths;
```

9/5
```
package body Shortest_Paths is
   function Shortest_Path
     (G : Graphs.Vector; Source : Node; Target : Node) return Paths.List
   is
      use Node_Maps, Paths, Graphs;
      Reached  : array (Node) of Boolean := (others => False);
      -- The set of nodes whose shortest distance to the source is known.
```

10/3
```
      Reached_From : array (Node) of Node;
      So_Far   : array (Node) of Distance := (others => Distance'Last);
      The_Path : Paths.List := Paths.Empty_List;
      Nearest_Distance : Distance;
      Next     : Node;
   begin
      So_Far(Source)  := 0.0;
```

```
      while not Reached(Target) loop                                          11/3
         Nearest_Distance := Distance'Last;
         -- Find closest node not reached yet, by iterating over all nodes.  12/3
         -- A more efficient algorithm uses a priority queue for this step.
         Next := Source;                                                      13/3
         for N in Node'First .. Node'Last loop
            if not Reached(N)
              and then So_Far(N) < Nearest_Distance then
                 Next := N;
                 Nearest_Distance := So_Far(N);
            end if;
         end loop;
         if Nearest_Distance = Distance'Last then                             14/3
            -- No next node found, graph is not connected
            return Paths.Empty_List;
         else                                                                 15/3
            Reached(Next) := True;
         end if;
         -- Update minimum distance to newly reachable nodes.                 16/3
         for E of G (Next) loop                                               17/3
            if not Reached(E.To) then
               Nearest_Distance := E.Length + So_Far(Next);
               if Nearest_Distance < So_Far(E.To) then                        18/3
                  Reached_From(E.To) := Next;
                  So_Far(E.To) := Nearest_Distance;
               end if;
            end if;
         end loop;
      end loop;
      -- Rebuild path from target to source.                                 19/3
      declare                                                                20/5
         N : Node := Target;
      begin
         Prepend (The_Path, N);
         while N /= Source loop
            N := Reached_From(N);
            Prepend (The_Path, N);
         end loop;
      end;
      return The_Path;                                                       21/3
   end;
end Shortest_Paths;
```

Note that the effect of the Constant_Indexing aspect (on type Vector) and the Implicit_Dereference aspect (on type Reference_Type) is that 22/3

```
G (Next)                                                                     23/3
```

is a convenient shorthand for 24/5

```
G.Constant_Reference (Next).Element.all                                      25/3
```

Similarly, the effect of the loop: 26/3

```
for E of G (Next) loop                                                       27/3
   if not Reached(E.To) then
      ...
   end if;
end loop;
```

28/3 is the same as:

29/4
```
for C in G (Next).Iterate loop
   declare
      E : Edge renames G (Next)(C);
   begin
      if not Reached(E.To) then
         ...
      end if;
   end;
end loop;
```

30/3 which is the same as:

31/4
```
declare
   L : Adjacency_Lists.List renames G (Next);
   C : Adjacency_Lists.Cursor := L.First;
begin
   while Has_Element (C) loop
      declare
         E : Edge renames L(C);
      begin
         if not Reached(E.To) then
            ...
         end if;
      end;
      C := L.Next (C);
   end loop;
end;
```

A.19 The Package Locales

1/3 A *locale* identifies a geopolitical place or region and its associated language, which can be used to determine other internationalization-related characteristics.

Static Semantics

2/3 The library package Locales has the following declaration:

3/5
```
package Ada.Locales
   with Preelaborate, Remote_Types is
```

4/4
```
   type Language_Code is new String (1 .. 3)
      with Dynamic_Predicate =>
         (for all E of Language_Code => E in 'a' .. 'z');
   type Country_Code is new String (1 .. 2)
      with Dynamic_Predicate =>
         (for all E of Country_Code  => E in 'A' .. 'Z');
```

5/3
```
   Language_Unknown : constant Language_Code := "und";
   Country_Unknown : constant Country_Code := "ZZ";
```

6/3
```
   function Language return Language_Code;
   function Country return Country_Code;
```

7/3
```
end Ada.Locales;
```

8/3 The *active locale* is the locale associated with the partition of the current task.

9/5 *This paragraph was deleted.*

10/5 *This paragraph was deleted.*

11/5 Function Language returns the code of the language associated with the active locale. If the Language_Code associated with the active locale cannot be determined from the environment, then

Language returns Language_Unknown. Otherwise, the result is a lower-case string representation of an ISO 639-3:2007 alpha-3 code that identifies a language.

Function Country returns the code of the country associated with the active locale. If the Country_Code associated with the active locale cannot be determined from the environment, then Country returns Country_Unknown. Otherwise, the result is an upper-case string representation of an ISO 3166-1:2020 alpha-2 code that identifies a country. 12/5

Annex B
(normative)
Interface to Other Languages

This Annex describes features for writing mixed-language programs. General interface support is presented first; then specific support for C, COBOL, and Fortran is defined, in terms of language interface packages for each of these languages. 1

Implementation Requirements

Support for interfacing to any foreign language is optional. However, an implementation shall not provide any optional aspect, attribute, library unit, or pragma having the same name as an aspect, attribute, library unit, or pragma (respectively) specified in the subclauses of this Annex unless the provided construct is either as specified in those subclauses or is more limited in capability than that required by those subclauses. A program that attempts to use an unsupported capability of this Annex shall either be identified by the implementation before run time or shall raise an exception at run time. 2/3

B.1 Interfacing Aspects

An *interfacing* aspect is a representation aspect that is one of the aspects Import, Export, Link_Name, External_Name, or Convention. 0.1/3

Specifying the Import aspect to have the value True is used to import an entity defined in a foreign language into an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada. In contrast, specifying the Export aspect to have the value True is used to export an Ada entity to a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language. The Import and Export aspects are intended primarily for objects and subprograms, although implementations are allowed to support other entities. The Link_Name and External_Name aspects are used to specify the link name and external name, respectively, to be used to identify imported or exported entities in the external environment. 1/3

The Convention aspect is used to indicate that an Ada entity should use the conventions of another language. It is intended primarily for types and "callback" subprograms. For example, "**with** Convention => Fortran" on the declaration of an array type Matrix implies that Matrix should be represented according to the conventions of the supported Fortran implementation, namely column-major order. 2/3

A pragma Linker_Options is used to specify the system linker parameters necessary when a given compilation unit is included in a partition. 3/5

Syntax

The form of a pragma Linker_Options is as follows: 4/3

Paragraphs 5 through 7 were moved to Annex J, "Obsolescent Features".

pragma Linker_Options(*string*_expression); 8

A pragma Linker_Options is allowed only at the place of a declarative_item. 9

This paragraph was deleted. 9.1/3

Name Resolution Rules

9.2/3 The Import and Export aspects are of type Boolean.

10/3 The Link_Name and External_Name aspects are of type String.

10.1/3 The expected type for the *string*_expression in pragma Linker_Options is String.

Legality Rules

11/3 The aspect Convention shall be specified by a *convention*_identifier which shall be the name of a *convention*. The convention names are implementation defined, except for certain language-defined ones, such as Ada and Intrinsic, as explained in 6.3.1, “Conformance Rules”. Additional convention names generally represent the calling conventions of foreign languages, language implementations, or specific run-time models. The convention of a callable entity is its *calling convention*.

12 If *L* is a *convention*_identifier for a language, then a type T is said to be *compatible with convention L*, (alternatively, is said to be an *L-compatible type*) if any of the following conditions are met:

13 • T is declared in a language interface package corresponding to *L* and is defined to be *L*-compatible (see B.3, B.3.1, B.3.2, B.4, B.5),

14/3 • Convention *L* has been specified for T, and T is *eligible for convention L*; that is:

14.1/4 • T is an enumeration type such that all internal codes (whether assigned by default or explicitly) are within an implementation-defined range that includes at least the range of values 0 .. 2**15–1;

15 • T is an array type with either an unconstrained or statically-constrained first subtype, and its component type is *L*-compatible,

16 • T is a record type that has no discriminants and that only has components with statically-constrained subtypes, and each component type is *L*-compatible,

17/3 • T is an access-to-object type, its designated type is *L*-compatible, and its designated subtype is not an unconstrained array subtype,

18 • T is an access-to-subprogram type, and its designated profile's parameter and result types are all *L*-compatible.

19 • T is derived from an *L*-compatible type,

19.1/5 • T is an anonymous access type, and T is eligible for convention *L*,

20 • The implementation permits T as an *L*-compatible type.

21/3 If the Convention aspect is specified for a type, then the type shall either be compatible with or eligible for the specified convention.

21.1/5 If convention *L* is specified for a type T, for each component of T that has an anonymous access type, the convention of the anonymous access type is *L*. If convention *L* is specified for an object that has an anonymous access type, the convention of the anonymous access type is *L*.

22/3 Notwithstanding any rule to the contrary, a declaration with a True Import aspect shall not have a completion.

23/3 An entity with a True Import aspect (or Export aspect) is said to be *imported* (respectively, *exported*). An entity shall not be both imported and exported.

24 The declaration of an imported object shall not include an explicit initialization expression. Default initializations are not performed.

The type of an imported or exported object shall be compatible with the specified Convention aspect, if any. 25/3

For an imported or exported subprogram, the result and parameter types shall each be compatible with the specified Convention aspect, if any. 26/3

The aspect_definition (if any) used to directly specify an Import, Export, External_Name, or Link_Name aspect shall be a static expression. The *string*_expression of a pragma Linker_Options shall be static. An External_Name or Link_Name aspect shall be specified only for an entity that is either imported or exported. 27/3

Static Semantics

Paragraphs 28 and 29 were deleted.

The Convention aspect represents the calling convention or representation convention of the entity. For an access-to-subprogram type, it represents the calling convention of designated subprograms. In addition: 30/3

- A True Import aspect indicates that the entity is defined externally (that is, outside the Ada program). This aspect is never inherited; if not directly specified, the Import aspect is False. 31/3
- A True Export aspect indicates that the entity is used externally. This aspect is never inherited; if not directly specified, the Export aspect is False. 32/3
- For an entity with a True Import or Export aspect, an external name, link name, or both may also be specified. 33/3

An *external name* is a string value for the name used by a foreign language program either for an entity that an Ada program imports, or for referring to an entity that an Ada program exports. 34

A *link name* is a string value for the name of an exported or imported entity, based on the conventions of the foreign language's compiler in interfacing with the system's linker tool. 35

The meaning of link names is implementation defined. If neither a link name nor the Address attribute of an imported or exported entity is specified, then a link name is chosen in an implementation-defined manner, based on the external name if one is specified. 36

Pragma Linker_Options has the effect of passing its string argument as a parameter to the system linker (if one exists), if the immediately enclosing compilation unit is included in the partition being linked. The interpretation of the string argument, and the way in which the string arguments from multiple Linker_Options pragmas are combined, is implementation defined. 37

Dynamic Semantics

Notwithstanding what this document says elsewhere, the elaboration of a declaration with a True Import aspect does not create the entity. Such an elaboration has no other effect than to allow the defining name to denote the external entity. 38/3

Erroneous Execution

It is the programmer's responsibility to ensure that the use of interfacing aspects does not violate Ada semantics; otherwise, program execution is erroneous. For example, passing an object with mode **in** to imported code that modifies it causes erroneous execution. Similarly, calling an imported subprogram that is not pure from a pure package causes erroneous execution. 38.1/5

Implementation Advice

39/3 If an implementation supports Export for a given language, then it should also allow the main subprogram to be written in that language. It should support some mechanism for invoking the elaboration of the Ada library units included in the system, and for invoking the finalization of the environment task. On typical systems, the recommended mechanism is to provide two subprograms whose link names are "adainit" and "adafinal". Adainit should contain the elaboration code for library units. Adafinal should contain the finalization code. These subprograms should have no effect the second and subsequent time they are called.

40/3 Automatic elaboration of preelaborated packages should be provided when specifying the Export aspect as True is supported.

41/5 For each supported convention *L* other than Intrinsic, an implementation should support specifying the Import and Export aspects for objects of *L*-compatible types and for subprograms, and the Convention aspect for *L*-eligible types and for subprograms, presuming the other language has corresponding features. Specifying the Convention aspect should be supported for enumeration types whose internal codes fall within the range 0 .. 2**15–1, but no recommendation is made for other scalar types.

42/5 NOTE 1 Implementations can place restrictions on interfacing aspects; for example, requiring each exported entity to be declared at the library level.

43/3 NOTE 2 The Convention aspect in combination with the Import aspect indicates the conventions for accessing external entities. It is possible that the actual entity is written in assembly language, but reflects the conventions of a particular language. For example, `with Convention => Ada` can be used to interface to an assembly language routine that obeys the Ada compiler's calling conventions.

44/5 NOTE 3 To obtain “call-back” to an Ada subprogram from a foreign language environment, the Convention aspect can be specified both for the access-to-subprogram type and the specific subprogram(s) to which 'Access is applied.

Paragraphs 45 and 46 were deleted.

47/5 NOTE 4 Machine code insertions can also be relevant for interfacing; see 13.8.

48/3 NOTE 5 If both External_Name and Link_Name are specified for a given entity, then the External_Name is ignored.

49/2 *This paragraph was deleted.*

Examples

50/4 *Example of interfacing aspects:*

51/3
```
package Fortran_Library is
  function Sqrt (X : Float) return Float
    with Import => True, Convention => Fortran;
  type Matrix is array (Natural range <>, Natural range <>) of Float
    with Convention => Fortran;
  function Invert (M : Matrix) return Matrix
    with Import => True, Convention => Fortran;
end Fortran_Library;
```

B.2 The Package Interfaces

Package Interfaces is the parent of several library packages that declare types and other entities useful for interfacing to foreign languages. It also contains some implementation-defined types that are useful across more than one language (in particular for interfacing to assembly language). 1

Static Semantics

The library package Interfaces has the following skeletal declaration: 2

```
package Interfaces                                                                    3/5
   with Pure is

   type Integer_n is range -2**(n-1) .. 2**(n-1) - 1;  --2's complement               4

   type Unsigned_n is mod 2**n;                                                       5

   function Shift_Left  (Value : Unsigned_n; Amount : Natural)                        6
      return Unsigned_n;
   function Shift_Right (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Shift_Right_Arithmetic (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Rotate_Left  (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   function Rotate_Right (Value : Unsigned_n; Amount : Natural)
      return Unsigned_n;
   ...
end Interfaces;
```

Implementation Requirements

An implementation shall provide the following declarations in the visible part of package Interfaces: 7

- Signed and modular integer types of *n* bits, if supported by the target architecture, for each *n* that is at least the size of a storage element and that is a factor of the word size. The names of these types are of the form Integer_*n* for the signed types, and Unsigned_*n* for the modular types; 8
- For each such modular type in Interfaces, shifting and rotating subprograms as specified in the declaration of Interfaces above. These subprograms are Intrinsic. They operate on a bit-by-bit basis, using the binary representation of the value of the operands to yield a binary representation for the result. The Amount parameter gives the number of bits by which to shift or rotate. For shifting, zero bits are shifted in, except in the case of Shift_Right_Arithmetic, where one bits are shifted in if Value is at least half the modulus. 9
- Floating point types corresponding to each floating point format fully supported by the hardware. 10

Implementation Permissions

An implementation may provide implementation-defined library units that are children of Interfaces, and may add declarations to the visible part of Interfaces in addition to the ones defined above. 11

A child package of package Interfaces with the name of a convention may be provided independently of whether the convention is supported by the Convention aspect and vice versa. Such a child package should contain any declarations that would be useful for interfacing to the language (implementation) represented by the convention. Any declarations useful for interfacing to any language on the given hardware architecture should be provided directly in Interfaces. 11.1/3

Implementation Advice

This paragraph was deleted. 12/2

13/3 An implementation supporting an interface to C, COBOL, or Fortran should provide the corresponding package or packages described in the following subclauses.

B.3 Interfacing with C and C++

1/4 The facilities relevant to interfacing with the C language and the corresponding subset of the C++ language are the package Interfaces.C and its children, and support for specifying the Convention aspect with *convention_*identifiers C, C_Pass_By_Copy, and any of the C_Variadic_*n* conventions described below.

2/3 The package Interfaces.C contains the basic types, constants, and subprograms that allow an Ada program to pass scalars and strings to C and C++ functions. When this subclause mentions a C entity, the reference also applies to the corresponding entity in C++.

Static Semantics

3 The library package Interfaces.C has the following declaration:

4/5
```
package Interfaces.C
   with Pure is
```

5
```
   -- Declarations based on C's <limits.h>
```

6
```
   CHAR_BIT  : constant := implementation-defined;   -- typically 8
   SCHAR_MIN : constant := implementation-defined;   -- typically –128
   SCHAR_MAX : constant := implementation-defined;   -- typically 127
   UCHAR_MAX : constant := implementation-defined;   -- typically 255
```

7
```
   -- Signed and Unsigned Integers
   type int   is range implementation-defined;
   type short is range implementation-defined;
   type long  is range implementation-defined;
```

8
```
   type signed_char is range SCHAR_MIN .. SCHAR_MAX;
   for signed_char'Size use CHAR_BIT;
```

9
```
   type unsigned       is mod implementation-defined;
   type unsigned_short is mod implementation-defined;
   type unsigned_long  is mod implementation-defined;
```

10
```
   type unsigned_char is mod (UCHAR_MAX+1);
   for unsigned_char'Size use CHAR_BIT;
```

11
```
   subtype plain_char is implementation-defined;
```

12
```
   type ptrdiff_t is range implementation-defined;
```

13
```
   type size_t is mod implementation-defined;
```

13.1/5
```
   -- Boolean Type
   type C_bool is new Boolean;
```

14
```
   -- Floating Point
```

15
```
   type C_float     is digits implementation-defined;
```

16
```
   type double      is digits implementation-defined;
```

17
```
   type long_double is digits implementation-defined;
```

18
```
   -- Characters and Strings
```

19
```
   type char is <implementation-defined character type>;
```

20/1
```
   nul : constant char := implementation-defined;
```

21
```
   function To_C   (Item : in Character) return char;
```

22
```
   function To_Ada (Item : in char) return Character;
```

23/3
```
   type char_array is array (size_t range <>) of aliased char
      with Pack;
   for char_array'Component_Size use CHAR_BIT;
```

```
function Is_Nul_Terminated (Item : in char_array) return Boolean;                    24
function To_C   (Item       : in String;                                             25
                 Append_Nul : in Boolean := True)
   return char_array;
function To_Ada (Item     : in char_array;                                           26
                 Trim_Nul : in Boolean := True)
   return String;
procedure To_C (Item       : in  String;                                            27
                Target     : out char_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);
procedure To_Ada (Item     : in  char_array;                                        28
                  Target   : out String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
-- Wide Character and Wide String                                                    29
type wchar_t is <implementation-defined character type>;                             30/1
wide_nul : constant wchar_t := implementation-defined;                               31/1
function To_C   (Item : in Wide_Character) return wchar_t;                           32
function To_Ada (Item : in wchar_t       ) return Wide_Character;
type wchar_array is array (size_t range <>) of aliased wchar_t                       33/3
   with Pack;
This paragraph was deleted.                                                          34/3
function Is_Nul_Terminated (Item : in wchar_array) return Boolean;                   35
function To_C   (Item       : in Wide_String;                                        36
                 Append_Nul : in Boolean := True)
   return wchar_array;
function To_Ada (Item     : in wchar_array;                                          37
                 Trim_Nul : in Boolean := True)
   return Wide_String;
procedure To_C (Item       : in  Wide_String;                                       38
                Target     : out wchar_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);
procedure To_Ada (Item     : in  wchar_array;                                       39
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
-- ISO/IEC 10646 compatible types.                                                   39.1/5
type char16_t is <implementation-defined character type>;                            39.2/2
char16_nul : constant char16_t := implementation-defined;                            39.3/2
function To_C (Item : in Wide_Character) return char16_t;                            39.4/2
function To_Ada (Item : in char16_t) return Wide_Character;
type char16_array is array (size_t range <>) of aliased char16_t                     39.5/3
   with Pack;
This paragraph was deleted.                                                          39.6/3
function Is_Nul_Terminated (Item : in char16_array) return Boolean;                  39.7/2
function To_C (Item       : in Wide_String;
               Append_Nul : in Boolean := True)
   return char16_array;
function To_Ada (Item     : in char16_array;                                         39.8/2
                 Trim_Nul : in Boolean := True)
   return Wide_String;
```

```
39.9/2          procedure To_C (Item       : in  Wide_String;
                                Target     : out char16_array;
                                Count      : out size_t;
                                Append_Nul : in  Boolean := True);
39.10/2         procedure To_Ada (Item     : in  char16_array;
                                  Target   : out Wide_String;
                                  Count    : out Natural;
                                  Trim_Nul : in  Boolean := True);
39.11/2         type char32_t is <implementation-defined character type>;
39.12/2         char32_nul : constant char32_t := implementation-defined;
39.13/2         function To_C (Item : in Wide_Wide_Character) return char32_t;
                function To_Ada (Item : in char32_t) return Wide_Wide_Character;
39.14/3         type char32_array is array (size_t range <>) of aliased char32_t
                   with Pack;
39.15/3      This paragraph was deleted.
39.16/2         function Is_Nul_Terminated (Item : in char32_array) return Boolean;
                function To_C (Item       : in Wide_Wide_String;
                               Append_Nul : in Boolean := True)
                   return char32_array;
39.17/2         function To_Ada (Item     : in char32_array;
                                 Trim_Nul : in Boolean := True)
                   return Wide_Wide_String;
39.18/2         procedure To_C (Item       : in  Wide_Wide_String;
                                Target     : out char32_array;
                                Count      : out size_t;
                                Append_Nul : in  Boolean := True);
39.19/2         procedure To_Ada (Item     : in  char32_array;
                                  Target   : out Wide_Wide_String;
                                  Count    : out Natural;
                                  Trim_Nul : in  Boolean := True);
40              Terminator_Error : exception;
41           end Interfaces.C;
```

42 Each of the types declared in Interfaces.C is C-compatible.

43/5 The types int, short, long, unsigned, ptrdiff_t, size_t, double, char, wchar_t, char16_t, and char32_t correspond respectively to the C types having the same names. The types signed_char, unsigned_short, unsigned_long, unsigned_char, C_bool, C_float, and long_double correspond respectively to the C types signed char, unsigned short, unsigned long, unsigned char, bool, float, and long double.

44 The type of the subtype plain_char is either signed_char or unsigned_char, depending on the C implementation.

```
45          function To_C   (Item : in Character) return char;
            function To_Ada (Item : in char     ) return Character;
```

46 The functions To_C and To_Ada map between the Ada type Character and the C type char.

```
47          function Is_Nul_Terminated (Item : in char_array) return Boolean;
```

48 The result of Is_Nul_Terminated is True if Item contains nul, and is False otherwise.

```
function To_C   (Item : in String;     Append_Nul : in Boolean := True)
   return char_array;

function To_Ada (Item : in char_array; Trim_Nul   : in Boolean := True)
   return String;
```

The result of To_C is a char_array value of length Item'Length (if Append_Nul is False) or Item'Length+1 (if Append_Nul is True). The lower bound is 0. For each component Item(I), the corresponding component in the result is To_C applied to Item(I). The value nul is appended if Append_Nul is True. If Append_Nul is False and Item'Length is 0, then To_C propagates Constraint_Error. 50/2

The result of To_Ada is a String whose length is Item'Length (if Trim_Nul is False) or the length of the slice of Item preceding the first nul (if Trim_Nul is True). The lower bound of the result is 1. If Trim_Nul is False, then for each component Item(I) the corresponding component in the result is To_Ada applied to Item(I). If Trim_Nul is True, then for each component Item(I) before the first nul the corresponding component in the result is To_Ada applied to Item(I). The function propagates Terminator_Error if Trim_Nul is True and Item does not contain nul. 51

```
procedure To_C (Item       : in  String;
                Target     : out char_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char_array;
                  Target   : out String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

For procedure To_C, each element of Item is converted (via the To_C function) to a char, which is assigned to the corresponding element of Target. If Append_Nul is True, nul is then assigned to the next element of Target. In either case, Count is set to the number of Target elements assigned. If Target is not long enough, Constraint_Error is propagated. 53

For procedure To_Ada, each element of Item (if Trim_Nul is False) or each element of Item preceding the first nul (if Trim_Nul is True) is converted (via the To_Ada function) to a Character, which is assigned to the corresponding element of Target. Count is set to the number of Target elements assigned. If Target is not long enough, Constraint_Error is propagated. If Trim_Nul is True and Item does not contain nul, then Terminator_Error is propagated. 54

```
function Is_Nul_Terminated (Item : in wchar_array) return Boolean;
```

The result of Is_Nul_Terminated is True if Item contains wide_nul, and is False otherwise. 56

```
function To_C   (Item : in Wide_Character) return wchar_t;
function To_Ada (Item : in wchar_t       ) return Wide_Character;
```

To_C and To_Ada provide the mappings between the Ada and C wide character types. 58

59
```
function To_C   (Item       : in Wide_String;
                 Append_Nul : in Boolean := True)
   return wchar_array;

function To_Ada (Item     : in wchar_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out wchar_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  wchar_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

60 The To_C and To_Ada subprograms that convert between Wide_String and wchar_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that wide_nul is used instead of nul.

60.1/2
```
function Is_Nul_Terminated (Item : in char16_array) return Boolean;
```

60.2/2 The result of Is_Nul_Terminated is True if Item contains char16_nul, and is False otherwise.

60.3/2
```
function To_C (Item : in Wide_Character) return char16_t;
function To_Ada (Item : in char16_t ) return Wide_Character;
```

60.4/2 To_C and To_Ada provide mappings between the Ada and C 16-bit character types.

60.5/2
```
function To_C (Item       : in Wide_String;
               Append_Nul : in Boolean := True)
   return char16_array;

function To_Ada (Item     : in char16_array;
                 Trim_Nul : in Boolean := True)
   return Wide_String;

procedure To_C (Item       : in  Wide_String;
                Target     : out char16_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char16_array;
                  Target   : out Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

60.6/2 The To_C and To_Ada subprograms that convert between Wide_String and char16_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char16_nul is used instead of nul.

60.7/2
```
function Is_Nul_Terminated (Item : in char32_array) return Boolean;
```

60.8/5 The result of Is_Nul_Terminated is True if Item contains char32_nul, and is False otherwise.

60.9/2
```
function To_C (Item : in Wide_Wide_Character) return char32_t;
function To_Ada (Item : in char32_t ) return Wide_Wide_Character;
```

60.10/2 To_C and To_Ada provide mappings between the Ada and C 32-bit character types.

```
function To_C (Item       : in Wide_Wide_String;                    60.11/2
               Append_Nul : in Boolean := True)
   return char32_array;

function To_Ada (Item     : in char32_array;
                 Trim_Nul : in Boolean := True)
   return Wide_Wide_String;

procedure To_C (Item       : in  Wide_Wide_String;
                Target     : out char32_array;
                Count      : out size_t;
                Append_Nul : in  Boolean := True);

procedure To_Ada (Item     : in  char32_array;
                  Target   : out Wide_Wide_String;
                  Count    : out Natural;
                  Trim_Nul : in  Boolean := True);
```

The To_C and To_Ada subprograms that convert between Wide_Wide_String and char32_array have analogous effects to the To_C and To_Ada subprograms that convert between String and char_array, except that char32_nul is used instead of nul. 60.12/2

The Convention aspect with *convention*_identifier C_Pass_By_Copy shall only be specified for a type. 60.13/3

The eligibility rules in B.1 do not apply to convention C_Pass_By_Copy. Instead, a type T is eligible for convention C_Pass_By_Copy if T is an unchecked union type or if T is a record type that has no discriminants and that only has components with statically constrained subtypes, and each component is C-compatible. 60.14/2

If a type is C_Pass_By_Copy-compatible, then it is also C-compatible. 60.15/3

The identifiers C_Variadic_0, C_Variadic_1, C_Variadic_2, and so on are *convention*_identifiers. These conventions are said to be *C_Variadic*. The convention C_Variadic_*n* is the calling convention for a variadic C function taking *n* fixed parameters and then a variable number of additional parameters. The C_Variadic_*n* convention shall only be specified as the convention aspect for a subprogram, or for an access-to-subprogram type, having at least *n* parameters. A type is compatible with a C_Variadic convention if and only if the type is C-compatible. 60.16/4

Implementation Requirements

An implementation shall support specifying aspect Convention with a C *convention*_identifier for a C-eligible type (see B.1). An implementation shall support specifying aspect Convention with a C_Pass_By_Copy *convention*_identifier for a C_Pass_By_Copy-eligible type. 61/3

Implementation Permissions

An implementation may provide additional declarations in the C interface packages. 62

An implementation is not required to support specifying the Convention aspect with *convention*_identifier C in the following cases: 62.1/5

- for a subprogram that has a parameter of an unconstrained array subtype, unless the Import aspect has the value True for the subprogram; 62.2/3
- for a function with an unconstrained array result subtype; 62.3/3
- for an object whose nominal subtype is an unconstrained array subtype. 62.4/3

Implementation Advice

The constants nul, wide_nul, char16_nul, and char32_nul should have a representation of zero. 62.5/3

63 An implementation should support the following interface correspondences between Ada and C.

64 • An Ada procedure corresponds to a void-returning C function.

65 • An Ada function corresponds to a non-void C function.

65.1/4 • An Ada enumeration type corresponds to a C enumeration type with corresponding enumeration literals having the same internal codes, provided the internal codes fall within the range of the C int type.

66 • An Ada **in** scalar parameter is passed as a scalar argument to a C function.

67 • An Ada **in** parameter of an access-to-object type with designated type T is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T.

68 • An Ada **access** T parameter, or an Ada **out** or **in out** parameter of an elementary type T, is passed as a t* argument to a C function, where t is the C type corresponding to the Ada type T. In the case of an elementary **out** or **in out** parameter, a pointer to a temporary copy is used to preserve by-copy semantics.

68.1/2 • An Ada parameter of a (record) type T of convention C_Pass_By_Copy, of mode **in**, is passed as a t argument to a C function, where t is the C struct corresponding to the Ada type T.

69/5 • An Ada parameter of a record type T, other than an **in** parameter of a type of convention C_Pass_By_Copy, is passed as a t* argument to a C function, with the const modifier if the Ada mode is **in**, where t is the C struct corresponding to the Ada type T.

70/5 • An Ada parameter of an array type with component type T is passed as a t* argument to a C function, with the const modifier if the Ada mode is **in**, where t is the C type corresponding to the Ada type T.

71 • An Ada parameter of an access-to-subprogram type is passed as a pointer to a C function whose prototype corresponds to the designated subprogram's specification.

71.1/3 • An Ada parameter of a private type is passed as specified for the full view of the type.

71.2/3 • The rules of correspondence given above for parameters of mode **in** also apply to the return object of a function.

71.3/5 An implementation should provide unsigned_long_long and long_long as 64-bit modular and signed integer types (respectively) in package Interfaces.C if the C implementation supports unsigned long long and long long as 64-bit types.

72 NOTE 1 Values of type char_array are not implicitly terminated with nul. If a char_array is to be passed as a parameter to an imported C function requiring nul termination, it is the programmer's responsibility to obtain this effect.

73 NOTE 2 To obtain the effect of C's sizeof(item_type), where Item_Type is the corresponding Ada type, evaluate the expression: size_t(Item_Type'Size/CHAR_BIT).

74/2 *This paragraph was deleted.*

75/4 NOTE 3 A variadic C function can correspond to several Ada subprograms, taking various specific numbers and types of parameters.

Examples

Example of using the Interfaces.C package: 76

```
--Calling the C Library Functions strcpy and printf                                    77/5
with Interfaces.C;
procedure Test is
   package C renames Interfaces.C;
   use type C.char_array;
   -- Call <string.h>strcpy:
   -- C definition of strcpy:  char *strcpy(char *s1, const char *s2);
   --    This function copies the string pointed to by s2 (including the terminating null character)
   --    into the array pointed to by s1. If copying takes place between objects that overlap,
   --    the behavior is undefined. The strcpy function returns the value of s1.

   -- Note: since the C function's return value is of no interest, the Ada interface is a procedure   78/3
   procedure Strcpy (Target : out C.char_array;
                     Source : in  C.char_array)
      with Import => True, Convention => C, External_Name => "strcpy";

   -- Call <sdtio.h>printf:                                                              79/5
   -- C definition of printf:  int printf ( const char * format, ... );
   --    This function writes the C string pointed by format to the standard output (stdout).
   --    If format includes format specifiers (subsequences beginning with %), the additional
   --    arguments following format are formatted and inserted in the resulting string
   --    replacing their respective specifiers. If the number of arguments does not match
   --    the number of format specifiers, or if the types of the arguments do not match
   --    the corresponding format specifier, the behaviour is undefined. On success, the
   --    printf function returns the total number of characters written to the standard output.
   --    If a writing error occurs, a negative number is returned.

   -- Note: since the C function's return value is of no interest, the Ada interface is a procedure   79.1/5
   procedure Printf (Format : in C.char_array;
                     Param1 : in C.char_array;
                     Param2 : in C.int)
      with Import => True, Convention => C_Variadic_1, External_Name =>
"printf";

   Chars1 :  C.char_array(1..20);                                                        80
   Chars2 :  C.char_array(1..20);

begin                                                                                    81
   Chars2(1..6) := "qwert" & C.nul;

   Strcpy(Chars1, Chars2);                                                               82

   -- Now Chars1(1..6) = "qwert" & C.Nul                                                 83

   Printf("The String=%s, Length=%d", Chars1, Chars1'Length);                            83.1/5

end Test;                                                                                84
```

B.3.1 The Package Interfaces.C.Strings

The package Interfaces.C.Strings declares types and subprograms allowing an Ada program to allocate, reference, update, and free C-style strings. In particular, the private type chars_ptr corresponds to a common use of “char *” in C programs, and an object of this type can be passed to a subprogram to which **with** `Import => True, Convention => C` has been specified, and for which “char *” is the type of the argument of the C function. 1/3

Static Semantics

The library package Interfaces.C.Strings has the following declaration: 2

```
package Interfaces.C.Strings                                                             3/5
   with Preelaborate, Nonblocking, Global => in out synchronized is

   type char_array_access is access all char_array;                                      4

   type chars_ptr is private                                                             5/5
      with Preelaborable_Initialization;
```

6/2
```
      type chars_ptr_array is array (size_t range <>) of aliased chars_ptr;
```
7
```
      Null_Ptr : constant chars_ptr;
```
8
```
      function To_Chars_Ptr (Item      : in char_array_access;
                             Nul_Check : in Boolean := False)
         return chars_ptr;
```
9
```
      function New_Char_Array (Chars   : in char_array) return chars_ptr;
```
10
```
      function New_String (Str : in String) return chars_ptr;
```
11
```
      procedure Free (Item : in out chars_ptr);
```
12
```
      Dereference_Error : exception;
```
13
```
      function Value (Item : in chars_ptr) return char_array;
```
14
```
      function Value (Item : in chars_ptr; Length : in size_t)
         return char_array;
```
15
```
      function Value (Item : in chars_ptr) return String;
```
16
```
      function Value (Item : in chars_ptr; Length : in size_t)
         return String;
```
17
```
      function Strlen (Item : in chars_ptr) return size_t;
```
18
```
      procedure Update (Item   : in chars_ptr;
                        Offset : in size_t;
                        Chars  : in char_array;
                        Check  : in Boolean := True);
```
19
```
      procedure Update (Item   : in chars_ptr;
                        Offset : in size_t;
                        Str    : in String;
                        Check  : in Boolean := True);
```
20
```
      Update_Error : exception;
```
21
```
   private
      ... -- not specified by the language
   end Interfaces.C.Strings;
```

22 The type chars_ptr is C-compatible and corresponds to the use of C's “char *” for a pointer to the first char in a char array terminated by nul. When an object of type chars_ptr is declared, its value is by default set to Null_Ptr, unless the object is imported (see B.1).

23
```
   function To_Chars_Ptr (Item      : in char_array_access;
                          Nul_Check : in Boolean := False)
      return chars_ptr;
```

24/3 If Item is **null**, then To_Chars_Ptr returns Null_Ptr. If Item is not **null**, Nul_Check is True, and Item.**all** does not contain nul, then the function propagates Terminator_Error; otherwise, To_Chars_Ptr performs a pointer conversion with no allocation of memory.

25
```
   function New_Char_Array (Chars   : in char_array) return chars_ptr;
```

26 This function returns a pointer to an allocated object initialized to Chars(Chars'First .. Index) & nul, where

27 • Index = Chars'Last if Chars does not contain nul, or

28 • Index is the smallest size_t value I such that Chars(I+1) = nul.

28.1 Storage_Error is propagated if the allocation fails.

29
```
   function New_String (Str : in String) return chars_ptr;
```

30 This function is equivalent to New_Char_Array(To_C(Str)).

```
procedure Free (Item : in out chars_ptr);
```

If Item is Null_Ptr, then Free has no effect. Otherwise, Free releases the storage occupied by Value(Item), and resets Item to Null_Ptr.

```
function Value (Item : in chars_ptr) return char_array;
```

If Item = Null_Ptr, then Value propagates Dereference_Error. Otherwise, Value returns the prefix of the array of chars pointed to by Item, up to and including the first nul. The lower bound of the result is 0. If Item does not point to a nul-terminated string, then execution of Value is erroneous.

```
function Value (Item : in chars_ptr; Length : in size_t)
   return char_array;
```

If Item = Null_Ptr, then Value propagates Dereference_Error. Otherwise, Value returns the shorter of two arrays, either the first Length chars pointed to by Item, or Value(Item). The lower bound of the result is 0. If Length is 0, then Value propagates Constraint_Error.

```
function Value (Item : in chars_ptr) return String;
```

Equivalent to To_Ada(Value(Item), Trim_Nul=>True).

```
function Value (Item : in chars_ptr; Length : in size_t)
   return String;
```

Equivalent to To_Ada(Value(Item, Length) & nul, Trim_Nul=>True).

```
function Strlen (Item : in chars_ptr) return size_t;
```

Returns *Val*'Length–1 where *Val* = Value(Item); propagates Dereference_Error if Item = Null_Ptr.

```
procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Chars  : in char_array;
                  Check  : Boolean := True);
```

If Item = Null_Ptr, then Update propagates Dereference_Error. Otherwise, this procedure updates the value pointed to by Item, starting at position Offset, using Chars as the data to be copied into the array. Overwriting the nul terminator, and skipping with the Offset past the nul terminator, are both prevented if Check is True, as follows:

- Let N = Strlen(Item). If Check is True, then:
  - If Offset+Chars'Length>N, propagate Update_Error.
  - Otherwise, overwrite the data in the array pointed to by Item, starting at the char at position Offset, with the data in Chars.
- If Check is False, then processing is as above, but with no check that Offset+Chars'Length>N.

```
procedure Update (Item   : in chars_ptr;
                  Offset : in size_t;
                  Str    : in String;
                  Check  : in Boolean := True);
```

Equivalent to Update(Item, Offset, To_C(Str, Append_Nul => False), Check).

Erroneous Execution

51 Execution of any of the following is erroneous if the Item parameter is not null_ptr and Item does not point to a nul-terminated array of chars.

52 • a Value function not taking a Length parameter,

53 • the Free procedure,

54 • the Strlen function.

55 Execution of Free(X) is also erroneous if the chars_ptr X was not returned by New_Char_Array or New_String.

56 Reading or updating a freed char_array is erroneous.

57 Execution of Update is erroneous if Check is False and a call with Check equal to True would have propagated Update_Error.

58/5 NOTE New_Char_Array and New_String can be implemented either through the allocation function from the C environment (“malloc”) or through Ada dynamic memory allocation (“new”). The key points are

59/5 • the returned value (a chars_ptr) is represented as a C “char *” so that it can be passed to C functions;

60/5 • the allocated object can be freed by the programmer via a call of Free, rather than by calling a C function.

B.3.2 The Generic Package Interfaces.C.Pointers

1 The generic package Interfaces.C.Pointers allows the Ada programmer to perform C-style operations on pointers. It includes an access type Pointer, Value functions that dereference a Pointer and deliver the designated array, several pointer arithmetic operations, and “copy” procedures that copy the contents of a source pointer into the array designated by a destination pointer. As in C, it treats an object Ptr of type Pointer as a pointer to the first element of an array, so that for example, adding 1 to Ptr yields a pointer to the second element of the array.

2/5 The generic allows two styles of usage: one in which the array is terminated by a special terminator element; and another in which the programmer keeps track of the length.

Static Semantics

3 The generic library package Interfaces.C.Pointers has the following declaration:

4/5
```
generic
   type Index is (<>);
   type Element is private;
   type Element_Array is array (Index range <>) of aliased Element;
   Default_Terminator : Element;
package Interfaces.C.Pointers
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

5
```
   type Pointer is access all Element;
```

6
```
   function Value(Ref        : in Pointer;
                  Terminator : in Element := Default_Terminator)
      return Element_Array;
```

7
```
   function Value(Ref    : in Pointer;
                  Length : in ptrdiff_t)
      return Element_Array;
```

8
```
   Pointer_Error : exception;
```

9
```
   -- C-style Pointer arithmetic
```

```
   function "+" (Left : in Pointer;   Right : in ptrdiff_t) return Pointer
      with Convention => Intrinsic;
   function "+" (Left : in ptrdiff_t; Right : in Pointer)   return Pointer
      with Convention => Intrinsic;
   function "-" (Left : in Pointer;   Right : in ptrdiff_t) return Pointer
      with Convention => Intrinsic;
   function "-" (Left : in Pointer;   Right : in Pointer) return ptrdiff_t
      with Convention => Intrinsic;
```

10/3

```
   procedure Increment (Ref : in out Pointer)
      with Convention => Intrinsic;
   procedure Decrement (Ref : in out Pointer)
      with Convention => Intrinsic;
```

11/3

This paragraph was deleted.

12/3

```
   function Virtual_Length (Ref        : in Pointer;
                            Terminator : in Element := Default_Terminator)
      return ptrdiff_t;
```

13

```
   procedure Copy_Terminated_Array
      (Source     : in Pointer;
       Target     : in Pointer;
       Limit      : in ptrdiff_t := ptrdiff_t'Last;
       Terminator : in Element :=  Default_Terminator);
```

14

```
   procedure Copy_Array (Source  : in Pointer;
                         Target  : in Pointer;
                         Length  : in ptrdiff_t);
end Interfaces.C.Pointers;
```

15

16

The type Pointer is C-compatible and corresponds to one use of C's “Element *”. An object of type Pointer is interpreted as a pointer to the initial Element in an Element_Array. Two styles are supported:

17

- Explicit termination of an array value with Default_Terminator (a special terminator value);

18

- Programmer-managed length, with Default_Terminator treated simply as a data element.

19

```
function Value(Ref        : in Pointer;
               Terminator : in Element := Default_Terminator)
   return Element_Array;
```

20

This function returns an Element_Array whose value is the array pointed to by Ref, up to and including the first Terminator; the lower bound of the array is Index'First. Interfaces.C.Strings.Dereference_Error is propagated if Ref is **null**.

21

```
function Value(Ref    : in Pointer;
               Length : in ptrdiff_t)
   return Element_Array;
```

22

This function returns an Element_Array comprising the first Length elements pointed to by Ref. The exception Interfaces.C.Strings.Dereference_Error is propagated if Ref is **null**.

23

The "+" and "–" functions perform arithmetic on Pointer values, based on the Size of the array elements. In each of these functions, Pointer_Error is propagated if a Pointer parameter is **null**.

24

```
procedure Increment (Ref : in out Pointer);
```

25

Equivalent to Ref := Ref+1.

26

```
procedure Decrement (Ref : in out Pointer);
```

27

Equivalent to Ref := Ref–1.

28

29
```
function Virtual_Length (Ref        : in Pointer;
                         Terminator : in Element := Default_Terminator)
   return ptrdiff_t;
```

30 Returns the number of Elements, up to the one just before the first Terminator, in Value(Ref, Terminator).

31
```
procedure Copy_Terminated_Array
   (Source     : in Pointer;
    Target     : in Pointer;
    Limit      : in ptrdiff_t := ptrdiff_t'Last;
    Terminator : in Element := Default_Terminator);
```

32 This procedure copies Value(Source, Terminator) into the array pointed to by Target; it stops either after Terminator has been copied, or the number of elements copied is Limit, whichever occurs first. Dereference_Error is propagated if either Source or Target is **null**.

33
```
procedure Copy_Array (Source  : in Pointer;
                      Target  : in Pointer;
                      Length  : in ptrdiff_t);
```

34 This procedure copies the first Length elements from the array pointed to by Source, into the array pointed to by Target. Dereference_Error is propagated if either Source or Target is **null**.

Erroneous Execution

35 It is erroneous to dereference a Pointer that does not designate an aliased Element.

36 Execution of Value(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator.

37 Execution of Value(Ref, Length) is erroneous if Ref does not designate an aliased Element in an Element_Array containing at least Length Elements between the designated Element and the end of the array, inclusive.

38 Execution of Virtual_Length(Ref, Terminator) is erroneous if Ref does not designate an aliased Element in an Element_Array terminated by Terminator.

39 Execution of Copy_Terminated_Array(Source, Target, Limit, Terminator) is erroneous in either of the following situations:

40 • Execution of both Value(Source, Terminator) and Value(Source, Limit) are erroneous, or

41 • Copying writes past the end of the array containing the Element designated by Target.

42 Execution of Copy_Array(Source, Target, Length) is erroneous if either Value(Source, Length) is erroneous, or copying writes past the end of the array containing the Element designated by Target.

43 NOTE To compose a Pointer from an Element_Array, use 'Access on the first element. For example (assuming appropriate instantiations):

44
```
Some_Array   : Element_Array(0..5) ;
Some_Pointer : Pointer := Some_Array(0)'Access;
```

Examples

Example of Interfaces.C.Pointers: 45

```
with Interfaces.C.Pointers;                                                    46
with Interfaces.C.Strings;
procedure Test_Pointers is
   package C renames Interfaces.C;
   package Char_Ptrs is
      new C.Pointers (Index              => C.size_t,
                      Element            => C.char,
                      Element_Array      => C.char_array,
                      Default_Terminator => C.nul);
   use type Char_Ptrs.Pointer;                                                 47
   subtype Char_Star is Char_Ptrs.Pointer;
   procedure Strcpy (Target_Ptr, Source_Ptr : Char_Star) is                    48
      Target_Temp_Ptr : Char_Star := Target_Ptr;
      Source_Temp_Ptr : Char_Star := Source_Ptr;
      Element : C.char;
   begin
      if Target_Temp_Ptr = null or Source_Temp_Ptr = null then
         raise C.Strings.Dereference_Error;
      end if;
      loop                                                                     49/1
         Element             := Source_Temp_Ptr.all;
         Target_Temp_Ptr.all := Element;
         exit when C."="(Element, C.nul);
         Char_Ptrs.Increment(Target_Temp_Ptr);
         Char_Ptrs.Increment(Source_Temp_Ptr);
      end loop;
   end Strcpy;
begin
   ...
end Test_Pointers;
```

B.3.3 Unchecked Union Types

Specifying aspect Unchecked_Union to have the value True defines an interface correspondence between a given discriminated type and some C union. The aspect requires that the associated type shall be given a representation that allocates no space for its discriminant(s). 1/3

Paragraphs 2 through 3 were moved to Annex J, "Obsolescent Features".

Static Semantics

For a discriminated record type having a variant_part, the following language-defined representation aspect may be specified: 3.1/3

Unchecked_Union 3.2/3

The type of aspect Unchecked_Union is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

Legality Rules

Paragraphs 4 and 5 were deleted.

A type for which aspect Unchecked_Union is True is called an *unchecked union type*. A subtype of an unchecked union type is defined to be an *unchecked union subtype*. An object of an unchecked union type is defined to be an *unchecked union object*. 6/3

All component subtypes of an unchecked union type shall be C-compatible. 7/2

8/2 If a component subtype of an unchecked union type is subject to a per-object constraint, then the component subtype shall be an unchecked union subtype.

9/5 Any name that denotes a discriminant of an object of an unchecked union type shall occur within the declarative region of the type or as the selector_name of an aggregate, and shall not occur within a record_representation_clause.

10/3 The type of a component declared in a variant_part of an unchecked union type shall not need finalization. In addition to the places where Legality Rules normally apply (see 12.3), this rule also applies in the private part of an instance of a generic unit. For an unchecked union type declared within the body of a generic unit, or within the body of any of its descendant library units, no part of the type of a component declared in a variant_part of the unchecked union type shall be of a formal private type or formal private extension declared within the formal part of the generic unit.

11/2 The completion of an incomplete or private type declaration having a known_discriminant_part shall not be an unchecked union type.

12/2 An unchecked union subtype shall only be passed as a generic actual parameter if the corresponding formal type has no known discriminants or is an unchecked union type.

Static Semantics

13/2 An unchecked union type is eligible for convention C.

14/2 All objects of an unchecked union type have the same size.

15/2 Discriminants of objects of an unchecked union type are of size zero.

16/2 Any check which would require reading a discriminant of an unchecked union object is suppressed (see 11.5). These checks include:

17/5 • The check performed when addressing a variant component (that is, a component that was declared in a variant part) of an unchecked union object that the object has this component (see 4.1.3).

18/2 • Any checks associated with a type or subtype conversion of a value of an unchecked union type (see 4.6). This includes, for example, the check associated with the implicit subtype conversion of an assignment statement.

19/2 • The subtype membership check associated with the evaluation of a qualified expression (see 4.7) or an uninitialized allocator (see 4.8).

Dynamic Semantics

20/6 A view of an unchecked union object (including a type conversion or function call) has *inferable discriminants* if it has a constrained nominal subtype, unless either the object is a component of an enclosing unchecked union object that is subject to a per-object constraint and the enclosing object lacks inferable discriminants, or the object is a view of an extension_aggregate, and the ancestor_part is an expression of an unchecked union subtype that lacks inferable discriminants.

21/2 An expression of an unchecked union type has inferable discriminants if it is either a name of an object with inferable discriminants or a qualified expression whose subtype_mark denotes a constrained subtype.

22/2 Program_Error is raised in the following cases:

23/2 • Evaluation of the predefined equality operator for an unchecked union type if either of the operands lacks inferable discriminants.

24/6 • *This paragraph was deleted.*

- Evaluation of an individual membership test if the subtype_mark (if any) denotes a constrained unchecked union subtype and the *tested*_simple_expression lacks inferable discriminants. 25/5
- Conversion from a derived unchecked union type to an unconstrained non-unchecked-union type if the operand of the conversion lacks inferable discriminants. 26/2
- Execution of the default implementation of the Write or Read attribute of an unchecked union type. 27/2
- Execution of the default implementation of the Output or Input attribute of an unchecked union type if the type lacks default discriminant values. 28/2

Paragraph 29 was deleted.

NOTE The use of an unchecked union to obtain the effect of an unchecked conversion results in erroneous execution (see 11.5). Execution of the following example is erroneous even if Float'Size = Integer'Size: 30/2

```
type T (Flag : Boolean := False) is                                    31/3
   record
       case Flag is
           when False =>
               F1 : Float := 0.0;
           when True =>
               F2 : Integer := 0;
       end case;
    end record
    with Unchecked_Union;
X : T;                                                                 32/2
Y : Integer := X.F2; -- erroneous
```

B.4 Interfacing with COBOL

The facilities relevant to interfacing with the COBOL language are the package Interfaces.COBOL and support for specifying the Convention aspect with *convention*_identifier COBOL. 1/3

The COBOL interface package supplies several sets of facilities: 2

- A set of types corresponding to the native COBOL types of the supported COBOL implementation (so-called “internal COBOL representations”), allowing Ada data to be passed as parameters to COBOL programs 3
- A set of types and constants reflecting external data representations such as can be found in files or databases, allowing COBOL-generated data to be read by an Ada program, and Ada-generated data to be read by COBOL programs 4/5
- A generic package for converting between an Ada decimal type value and either an internal or external COBOL representation 5

Static Semantics

The library package Interfaces.COBOL has the following declaration: 6

```
package Interfaces.COBOL                                               7/5
   with Preelaborate, Nonblocking, Global => in out synchronized is
   -- Types and operations for internal data representations           8
      type Floating      is digits implementation-defined;             9
      type Long_Floating is digits implementation-defined;
      type Binary      is range implementation-defined;                10
      type Long_Binary is range implementation-defined;
      Max_Digits_Binary      : constant := implementation-defined;     11
      Max_Digits_Long_Binary : constant := implementation-defined;
```

```
12/3        type Decimal_Element  is mod implementation-defined;
            type Packed_Decimal is array (Positive range <>) of Decimal_Element
               with Pack;
13          type COBOL_Character is implementation-defined character type;
14          Ada_To_COBOL : array (Character) of COBOL_Character := implementation-defined;
15          COBOL_To_Ada : array (COBOL_Character) of Character := implementation-defined;
16/3        type Alphanumeric is array (Positive range <>) of COBOL_Character
               with Pack;
17          function To_COBOL (Item : in String) return Alphanumeric;
            function To_Ada   (Item : in Alphanumeric) return String;
18          procedure To_COBOL (Item       : in String;
                                Target     : out Alphanumeric;
                                Last       : out Natural);
19          procedure To_Ada (Item     : in Alphanumeric;
                              Target   : out String;
                              Last     : out Natural);
20/3        type Numeric is array (Positive range <>) of COBOL_Character
               with Pack;
21       -- Formats for COBOL data representations
22          type Display_Format is private;
23          Unsigned             : constant Display_Format;
            Leading_Separate     : constant Display_Format;
            Trailing_Separate    : constant Display_Format;
            Leading_Nonseparate  : constant Display_Format;
            Trailing_Nonseparate : constant Display_Format;
24          type Binary_Format is private;
25          High_Order_First  : constant Binary_Format;
            Low_Order_First   : constant Binary_Format;
            Native_Binary     : constant Binary_Format;
26          type Packed_Format is private;
27          Packed_Unsigned   : constant Packed_Format;
            Packed_Signed     : constant Packed_Format;
28       -- Types for external representation of COBOL binary data
29/3        type Byte is mod 2**COBOL_Character'Size;
            type Byte_Array is array (Positive range <>) of Byte
               with Pack;
30          Conversion_Error : exception;
31          generic
               type Num is delta <> digits <>;
            package Decimal_Conversions is
32             -- Display Formats: data values are represented as Numeric
33             function Valid (Item   : in Numeric;
                               Format : in Display_Format) return Boolean;
34             function Length (Format : in Display_Format) return Natural;
35             function To_Decimal (Item   : in Numeric;
                                    Format : in Display_Format) return Num;
36             function To_Display (Item   : in Num;
                                    Format : in Display_Format) return Numeric;
37             -- Packed Formats: data values are represented as Packed_Decimal
38             function Valid (Item   : in Packed_Decimal;
                               Format : in Packed_Format) return Boolean;
39             function Length (Format : in Packed_Format) return Natural;
```

```
      function To_Decimal (Item   : in Packed_Decimal;                    40
                           Format : in Packed_Format) return Num;
      function To_Packed (Item   : in Num;                                41
                          Format : in Packed_Format) return Packed_Decimal;
      -- Binary Formats: external data values are represented as Byte_Array   42
      function Valid (Item   : in Byte_Array;                             43
                      Format : in Binary_Format) return Boolean;
      function Length (Format : in Binary_Format) return Natural;         44
      function To_Decimal (Item   : in Byte_Array;
                           Format : in Binary_Format) return Num;
      function To_Binary (Item   : in Num;                                45
                          Format : in Binary_Format) return Byte_Array;
      -- Internal Binary formats: data values are of type Binary or Long_Binary   46
      function To_Decimal (Item : in Binary)      return Num;             47
      function To_Decimal (Item : in Long_Binary) return Num;
      function To_Binary      (Item : in Num)  return Binary;             48
      function To_Long_Binary (Item : in Num)  return Long_Binary;
   end Decimal_Conversions;                                               49
private                                                                   50
   ... -- not specified by the language
end Interfaces.COBOL;
```

Each of the types in Interfaces.COBOL is COBOL-compatible. 51

The types Floating and Long_Floating correspond to the native types in COBOL for data items with computational usage implemented by floating point. The types Binary and Long_Binary correspond to the native types in COBOL for data items with binary usage, or with computational usage implemented by binary. 52

Max_Digits_Binary is the largest number of decimal digits in a numeric value that is represented as Binary. Max_Digits_Long_Binary is the largest number of decimal digits in a numeric value that is represented as Long_Binary. 53

The type Packed_Decimal corresponds to COBOL's packed-decimal usage. 54

The type COBOL_Character defines the run-time character set used in the COBOL implementation. Ada_To_COBOL and COBOL_To_Ada are the mappings between the Ada and COBOL run-time character sets. 55

Type Alphanumeric corresponds to COBOL's alphanumeric data category. 56

Each of the functions To_COBOL and To_Ada converts its parameter based on the mappings Ada_To_COBOL and COBOL_To_Ada, respectively. The length of the result for each is the length of the parameter, and the lower bound of the result is 1. Each component of the result is obtained by applying the relevant mapping to the corresponding component of the parameter. 57

Each of the procedures To_COBOL and To_Ada copies converted elements from Item to Target, using the appropriate mapping (Ada_To_COBOL or COBOL_To_Ada, respectively). The index in Target of the last element assigned is returned in Last (0 if Item is a null array). If Item'Length exceeds Target'Length, Constraint_Error is propagated. 58

Type Numeric corresponds to COBOL's numeric data category with display usage. 59

60 The types Display_Format, Binary_Format, and Packed_Format are used in conversions between Ada decimal type values and COBOL internal or external data representations. The value of the constant Native_Binary is either High_Order_First or Low_Order_First, depending on the implementation.

61
```
function Valid (Item   : in Numeric;
                Format : in Display_Format) return Boolean;
```

62 The function Valid checks that the Item parameter has a value consistent with the value of Format. If the value of Format is other than Unsigned, Leading_Separate, and Trailing_Separate, the effect is implementation defined. If Format does have one of these values, the following rules apply:

63/3 • Format=Unsigned: if Item comprises one or more decimal digit characters, then Valid returns True, else it returns False.

64/1 • Format=Leading_Separate: if Item comprises a single occurrence of the plus or minus sign character, and then one or more decimal digit characters, then Valid returns True, else it returns False.

65/1 • Format=Trailing_Separate: if Item comprises one or more decimal digit characters and finally a plus or minus sign character, then Valid returns True, else it returns False.

66
```
function Length (Format : in Display_Format) return Natural;
```

67 The Length function returns the minimal length of a Numeric value sufficient to hold any value of type Num when represented as Format.

68
```
function To_Decimal (Item   : in Numeric;
                     Format : in Display_Format) return Num;
```

69 Produces a value of type Num corresponding to Item as represented by Format. The number of digits after the assumed radix point in Item is Num'Scale. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

70
```
function To_Display (Item   : in Num;
                     Format : in Display_Format) return Numeric;
```

71/1 This function returns the Numeric value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Unsigned.

72
```
function Valid (Item   : in Packed_Decimal;
                Format : in Packed_Format) return Boolean;
```

73 This function returns True if Item has a value consistent with Format, and False otherwise. The rules for the formation of Packed_Decimal values are implementation defined.

74
```
function Length (Format : in Packed_Format) return Natural;
```

75 This function returns the minimal length of a Packed_Decimal value sufficient to hold any value of type Num when represented as Format.

76
```
function To_Decimal (Item   : in Packed_Decimal;
                     Format : in Packed_Format) return Num;
```

77 Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num.

```
function To_Packed (Item   : in Num;                                    78
                    Format : in Packed_Format) return Packed_Decimal;
```

This function returns the Packed_Decimal value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. Conversion_Error is propagated if Num is negative and Format is Packed_Unsigned. 79/1

```
function Valid (Item   : in Byte_Array;                                 80
                Format : in Binary_Format) return Boolean;
```

This function returns True if Item has a value consistent with Format, and False otherwise. 81

```
function Length (Format : in Binary_Format) return Natural;             82
```

This function returns the minimal length of a Byte_Array value sufficient to hold any value of type Num when represented as Format. 83

```
function To_Decimal (Item   : in Byte_Array;                            84
                     Format : in Binary_Format) return Num;
```

Produces a value of type Num corresponding to Item as represented by Format. Num'Scale is the number of digits after the assumed radix point in Item. Conversion_Error is propagated if the value represented by Item is outside the range of Num. 85

```
function To_Binary (Item   : in Num;                                    86
                    Format : in Binary_Format) return Byte_Array;
```

This function returns the Byte_Array value for Item, represented in accordance with Format. The length of the returned value is Length(Format), and the lower bound is 1. 87/1

```
function To_Decimal (Item : in Binary)      return Num;                 88
function To_Decimal (Item : in Long_Binary) return Num;
```

These functions convert from COBOL binary format to a corresponding value of the decimal type Num. Conversion_Error is propagated if Item is too large for Num. 89

```
function To_Binary      (Item : in Num)  return Binary;                 90
function To_Long_Binary (Item : in Num)  return Long_Binary;
```

These functions convert from Ada decimal to COBOL binary format. Conversion_Error is propagated if the value of Item is too large to be represented in the result type. 91

Implementation Requirements

An implementation shall support specifying aspect Convention with a COBOL *convention*_identifier for a COBOL-eligible type (see B.1). 92/3

Implementation Permissions

An implementation may provide additional constants of the private types Display_Format, Binary_Format, or Packed_Format. 93

An implementation may provide further floating point and integer types in Interfaces.COBOL to match additional native COBOL types, and may also supply corresponding conversion functions in the generic package Decimal_Conversions. 94

Implementation Advice

An Ada implementation should support the following interface correspondences between Ada and COBOL. 95

96 • An Ada **access** T parameter is passed as a “BY REFERENCE” data item of the COBOL type corresponding to T.

97 • An Ada **in** scalar parameter is passed as a “BY CONTENT” data item of the corresponding COBOL type.

98 • Any other Ada parameter is passed as a “BY REFERENCE” data item of the COBOL type corresponding to the Ada parameter type; for scalars, a local copy is used if necessary to ensure by-copy semantics.

99/3 NOTE 1 An implementation is not required to support specifying aspect Convention for access types, nor is it required to support specifying aspects Import, Export, or Convention for functions.

100/5 NOTE 2 If an Ada subprogram is exported to COBOL, then a call from COBOL call can specify either “BY CONTENT” or “BY REFERENCE”.

Examples

101 *Examples of Interfaces.COBOL:*

102
```
with Interfaces.COBOL;
procedure Test_Call is
```

103
```
   -- Calling a foreign COBOL program
   -- Assume that a COBOL program PROG has the following declaration
   --  in its LINKAGE section:
   --  01 Parameter-Area
   --     05 NAME   PIC X(20).
   --     05 SSN    PIC X(9).
   --     05 SALARY PIC 99999V99 USAGE COMP.
   -- The effect of PROG is to update SALARY based on some algorithm
```

104
```
   package COBOL renames Interfaces.COBOL;
```

105
```
   type Salary_Type is delta 0.01 digits 7;
```

106/3
```
   type COBOL_Record is
      record
         Name   : COBOL.Numeric(1..20);
         SSN    : COBOL.Numeric(1..9);
         Salary : COBOL.Binary;  -- Assume Binary = 32 bits
      end record
      with Convention => COBOL;
```

107/3
```
   procedure Prog (Item : in out COBOL_Record)
      with Import => True, Convention => COBOL;
```

108
```
   package Salary_Conversions is
      new COBOL.Decimal_Conversions(Salary_Type);
```

109
```
   Some_Salary : Salary_Type := 12_345.67;
   Some_Record : COBOL_Record :=
      (Name   => "Johnson, John       ",
       SSN    => "111223333",
       Salary => Salary_Conversions.To_Binary(Some_Salary));
```

110
```
begin
   Prog (Some_Record);
   ...
end Test_Call;
```

111
```
with Interfaces.COBOL;
with COBOL_Sequential_IO; -- Assumed to be supplied by implementation
procedure Test_External_Formats is
```

```
-- Using data created by a COBOL program                                    112/5
-- Assume that a COBOL program has created a sequential file with
--  the following record structure, and that we want
--  process the records in an Ada program
--  01 EMPLOYEE-RECORD
--     05 NAME    PIC X(20).
--     05 SSN     PIC X(9).
--     05 SALARY  PIC 99999V99 USAGE COMP.
--     05 ADJUST  PIC S999V999 SIGN LEADING SEPARATE.
-- The COMP data is binary (32 bits), high-order byte first

package COBOL renames Interfaces.COBOL;                                     113

type Salary_Type      is delta 0.01  digits 7;                              114
type Adjustments_Type is delta 0.001 digits 6;

type COBOL_Employee_Record_Type is  -- External representation              115/3
   record
      Name    : COBOL.Alphanumeric(1..20);
      SSN     : COBOL.Alphanumeric(1..9);
      Salary  : COBOL.Byte_Array(1..4);
      Adjust  : COBOL.Numeric(1..7);  -- Sign and 6 digits
   end record
   with Convention => COBOL;

package COBOL_Employee_IO is                                                116
   new COBOL_Sequential_IO(COBOL_Employee_Record_Type);
use COBOL_Employee_IO;

COBOL_File : File_Type;                                                     117

type Ada_Employee_Record_Type is  -- Internal representation                118
   record
      Name    : String(1..20);
      SSN     : String(1..9);
      Salary  : Salary_Type;
      Adjust  : Adjustments_Type;
   end record;

COBOL_Record : COBOL_Employee_Record_Type;                                  119
Ada_Record   : Ada_Employee_Record_Type;

package Salary_Conversions is                                               120
   new COBOL.Decimal_Conversions(Salary_Type);
use Salary_Conversions;

package Adjustments_Conversions is                                          121
   new COBOL.Decimal_Conversions(Adjustments_Type);
use Adjustments_Conversions;

begin                                                                       122
   Open (COBOL_File, Name => "Some_File");

   loop                                                                     123
     Read (COBOL_File, COBOL_Record);

     Ada_Record.Name := COBOL.To_Ada(COBOL_Record.Name);                    124/5
     Ada_Record.SSN  := COBOL.To_Ada(COBOL_Record.SSN);
     Ada_Record.Salary :=
        To_Decimal(COBOL_Record.Salary, COBOL.High_Order_First);
     Ada_Record.Adjust :=
        To_Decimal(COBOL_Record.Adjust, COBOL.Leading_Separate);
     ... -- Process Ada_Record
   end loop;
exception
   when End_Error => ...
end Test_External_Formats;
```

B.5 Interfacing with Fortran

1/3 The facilities relevant to interfacing with the Fortran language are the package Interfaces.Fortran and support for specifying the Convention aspect with *convention*_identifier Fortran.

2 The package Interfaces.Fortran defines Ada types whose representations are identical to the default representations of the Fortran intrinsic types Integer, Real, Double Precision, Complex, Logical, and Character in a supported Fortran implementation. These Ada types can therefore be used to pass objects between Ada and Fortran programs.

Static Semantics

3 The library package Interfaces.Fortran has the following declaration:

4/5
```
with Ada.Numerics.Generic_Complex_Types;  -- see G.1.1
pragma Elaborate_All(Ada.Numerics.Generic_Complex_Types);
package Interfaces.Fortran
   with Pure is

   type Fortran_Integer is range implementation-defined;

   type Real             is digits implementation-defined;
   type Double_Precision is digits implementation-defined;

   type Logical is new Boolean;

   package Single_Precision_Complex_Types is
      new Ada.Numerics.Generic_Complex_Types (Real);

   type Complex is new Single_Precision_Complex_Types.Complex;

   subtype Imaginary is Single_Precision_Complex_Types.Imaginary;
   i : Imaginary renames Single_Precision_Complex_Types.i;
   j : Imaginary renames Single_Precision_Complex_Types.j;

10.1/5   package Double_Precision_Complex_Types is
      new Ada.Numerics.Generic_Complex_Types (Double_Precision);

10.2/5   type Double_Complex is new Double_Precision_Complex_Types.Complex;

10.3/5   subtype Double_Imaginary is Double_Precision_Complex_Types.Imaginary;

   type Character_Set is implementation-defined character type;

12/3   type Fortran_Character is array (Positive range <>) of Character_Set
      with Pack;

   function To_Fortran (Item : in Character) return Character_Set;
   function To_Ada (Item : in Character_Set) return Character;

   function To_Fortran (Item : in String) return Fortran_Character;
   function To_Ada     (Item : in Fortran_Character) return String;

   procedure To_Fortran (Item       : in String;
                         Target     : out Fortran_Character;
                         Last       : out Natural);

   procedure To_Ada (Item     : in Fortran_Character;
                     Target   : out String;
                     Last     : out Natural);

end Interfaces.Fortran;
```

18/5 The types Fortran_Integer, Real, Double_Precision, Logical, Complex, Double_Complex, Character_Set, and Fortran_Character are Fortran-compatible.

19 The To_Fortran and To_Ada functions map between the Ada type Character and the Fortran type Character_Set, and also between the Ada type String and the Fortran type Fortran_Character. The To_Fortran and To_Ada procedures have analogous effects to the string conversion subprograms found in Interfaces.COBOL.

Implementation Requirements

An implementation shall support specifying aspect Convention with a Fortran *convention_*identifier for a Fortran-eligible type (see B.1). 20/3

Implementation Permissions

An implementation may add additional declarations to the Fortran interface packages. For example, declarations are permitted for the character types corresponding to Fortran character kinds 'ascii' and 'iso_10646', which in turn correspond to ISO/IEC 646:1991 and to UCS-4 as specified in ISO/IEC 10646:2020. 21/5

Implementation Advice

An Ada implementation should support the following interface correspondences between Ada and Fortran: 22

- An Ada procedure corresponds to a Fortran subroutine. 23
- An Ada function corresponds to a Fortran function. 24
- An Ada parameter of an elementary, array, or record type T is passed as a T_F argument to a Fortran procedure, where T_F is the Fortran type corresponding to the Ada type T, and where the INTENT attribute of the corresponding dummy argument matches the Ada formal parameter mode; the Fortran implementation's parameter passing conventions are used. For elementary types, a local copy is used if necessary to ensure by-copy semantics. 25
- An Ada parameter of an access-to-subprogram type is passed as a reference to a Fortran procedure whose interface corresponds to the designated subprogram's specification. 26

NOTE 1 An object of a Fortran-compatible record type, declared in a library package or subprogram, can correspond to a Fortran common block; the type also corresponds to a Fortran “derived type”. 27

NOTE 2 For Fortran facilities not addressed by this subclause, consider using the Fortran to C interoperability features defined in ISO/IEC 1594-1:2018 along with the C interfacing features defined in B.3. 28/5

Examples

Example of Interfaces.Fortran: 29

```
with Interfaces.Fortran;                                                    30
use Interfaces.Fortran;
procedure Ada_Application is

   type Fortran_Matrix is                                                   31/5
      array (Fortran_Integer range <>,
             Fortran_Integer range <>) of Double_Precision
      with Convention => Fortran;                  -- stored in Fortran's
                                                   -- column-major order
   procedure Invert (Rank : in Fortran_Integer; X : in out Fortran_Matrix)
      with Import => True, Convention => Fortran; -- a Fortran subroutine

   Rank      : constant Fortran_Integer := 100;                             32
   My_Matrix : Fortran_Matrix (1 .. Rank, 1 .. Rank);

   Precision: constant := 6;                                                33/5
   type Standard_Deviation is digits Precision
      with Convention => Fortran;
   Deviation : Standard_Deviation;
      -- Declarations to match the following Fortran declarations:
      --   integer, parameter :: precision = selected_real_kind(p=6)
      --   real(precision) :: deviation

begin                                                                       34
```

```
35          ...
            My_Matrix := ...;
            ...
            Invert (Rank, My_Matrix);
            ...
36/5        Deviation := ...;
            ...
37      end Ada_Application;
```

Annex C
(normative)
Systems Programming

The Systems Programming Annex specifies additional capabilities provided for low-level programming. These capabilities are also required in many real-time, embedded, distributed, and information systems. 1

C.1 Access to Machine Operations

This subclause specifies rules regarding access to machine instructions from within an Ada program. 1/3

Implementation Requirements

The implementation shall support machine code insertions (see 13.8) or intrinsic subprograms (see 6.3.1) (or both). The implementation shall allow the use of Ada entities as operands for such machine code insertions or intrinsic subprograms. 2/5

Implementation Advice

The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment, including privileged instructions, if any. 3

The support for interfacing aspects (see Annex B) should include interface to assembler; the default assembler should be associated with the convention identifier Assembler. 4/3

If an entity is exported to assembly language, then the implementation should allocate it at an addressable location, and should ensure that it is retained by the linking process, even if not otherwise referenced from the Ada code. The implementation should assume that any call to a machine code or assembler subprogram is allowed to read or update every object that is specified as exported. 5

Documentation Requirements

The implementation shall document the overhead associated with calling machine-code or intrinsic subprograms, as compared to a fully-inlined call, and to a regular out-of-line call. 6

The implementation shall document the types of the package System.Machine_Code usable for machine code insertions, and the attributes to be used in machine code insertions for references to Ada entities. 7

The implementation shall document the subprogram calling conventions associated with the convention identifiers available for use with the Convention aspect (Ada and Assembler, at a minimum), including register saving, exception propagation, parameter passing, and function value returning. 8/3

For exported and imported subprograms, the implementation shall document the mapping between the Link_Name string, if specified, or the Ada designator, if not, and the external link name used for such a subprogram. 9

Implementation Advice

The implementation should ensure that little or no overhead is associated with calling intrinsic and machine-code subprograms. 10

11 It is recommended that intrinsic subprograms be provided for convenient access to any machine operations that provide special capabilities or efficiency and that are not otherwise available through the language constructs. Examples of such instructions include:

12/5 • Atomic read-modify-write operations — for example, test and set, compare and swap, decrement and test, enqueue/dequeue.

13/5 • Standard numeric functions — for example, *sin*, *log*.

14/5 • String manipulation operations — for example, translate and test.

15/5 • Vector operations — for example, compare vector against thresholds.

16 • Direct operations on I/O ports.

C.2 Required Representation Support

1/3 This subclause specifies minimal requirements on the support for representation items and related features.

Implementation Requirements

2 The implementation shall support at least the functionality defined by the recommended levels of support in Clause 13.

C.3 Interrupt Support

1/3 This subclause specifies the language-defined model for hardware interrupts in addition to mechanisms for handling interrupts.

Dynamic Semantics

2 An *interrupt* represents a class of events that are detected by the hardware or the system software. Interrupts are said to occur. An *occurrence* of an interrupt is separable into generation and delivery. *Generation* of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program. *Delivery* is the action that invokes part of the program as response to the interrupt occurrence. Between generation and delivery, the interrupt occurrence (or interrupt) is *pending*. Some or all interrupts may be *blocked*. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Certain interrupts are *reserved*. The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other implementation-defined means. Program units can be connected to nonreserved interrupts. While connected, the program unit is said to be *attached* to that interrupt. The execution of that program unit, the *interrupt handler*, is invoked upon delivery of the interrupt occurrence.

3 While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt. While the handler executes, the corresponding interrupt is blocked.

4 While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined.

5 Each interrupt has a *default treatment* which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is implementation defined, as is the method (if one exists) for configuring the default treatments for interrupts.

An interrupt is delivered to the handler (or default treatment) that is in effect for that interrupt at the time of delivery. 6

An exception propagated from a handler that is invoked by an interrupt has no effect. 7

If the Ceiling_Locking policy (see D.3) is in effect, the interrupt handler executes with the active priority that is the ceiling priority of the corresponding protected object. 8

Implementation Requirements

The implementation shall provide a mechanism to determine the minimum stack space that is necessary for each interrupt handler and to reserve that space for the execution of the handler. This space should accommodate nested invocations of the handler where the system permits this. 9/5

If the hardware or the underlying system holds pending interrupt occurrences, the implementation shall provide for later delivery of these occurrences to the program. 10

If the Ceiling_Locking policy is not in effect, the implementation shall provide means for the application to specify whether interrupts are to be blocked during protected actions. 11

Documentation Requirements

The implementation shall document the following items: 12

1. For each interrupt, which interrupts are blocked from delivery when a handler attached to that interrupt executes (either as a result of an interrupt delivery or of an ordinary call on a procedure of the corresponding protected object). 13
2. Any interrupts that cannot be blocked, and the effect of attaching handlers to such interrupts, if this is permitted. 14
3. Which run-time stack an interrupt handler uses when it executes as a result of an interrupt delivery; if this is configurable, what is the mechanism to do so; how to specify how much space to reserve on that stack. 15
4. Any implementation- or hardware-specific activity that happens before a user-defined interrupt handler gets control (for example, reading device registers, acknowledging devices). 16/5
5. Any timing or other limitations imposed on the execution of interrupt handlers. 17
6. The state (blocked/unblocked) of the nonreserved interrupts when the program starts; if some interrupts are unblocked, what is the mechanism a program can use to protect itself before it can attach the corresponding handlers. 18
7. Whether the interrupted task is allowed to resume execution before the interrupt handler returns. 19
8. The treatment of interrupt occurrences that are generated while the interrupt is blocked; that is, whether one or more occurrences are held for later delivery, or all are lost. 20/5
9. Whether predefined or implementation-defined exceptions are raised as a result of the occurrence of any interrupt, and the mapping between the machine interrupts (or traps) and the predefined exceptions. 21
10. On a multi-processor, the rules governing the delivery of an interrupt to a particular processor. 22

Implementation Permissions

If the underlying system or hardware does not allow interrupts to be blocked, then no blocking is required as part of the execution of subprograms of a protected object for which one of its subprograms is an interrupt handler. 23/2

24 In a multi-processor with more than one interrupt subsystem, it is implementation defined whether (and how) interrupt sources from separate subsystems share the same Interrupt_Id type (see C.3.2). In particular, the meaning of a blocked or pending interrupt may then be applicable to one processor only.

25 Implementations are allowed to impose timing or other limitations on the execution of interrupt handlers.

26/3 Other forms of handlers are allowed to be supported, in which case the rules of this subclause should be adhered to.

27 The active priority of the execution of an interrupt handler is allowed to vary from one occurrence of the same interrupt to another.

Implementation Advice

28/2 If the Ceiling_Locking policy is not in effect, the implementation should provide means for the application to specify which interrupts are to be blocked during protected actions, if the underlying system allows for finer-grained control of interrupt blocking.

29/5 NOTE 1 The default treatment for an interrupt can be to keep the interrupt pending or to deliver it to an implementation-defined handler. Examples of actions that an implementation-defined handler can perform include aborting the partition, ignoring (that is, discarding occurrences of) the interrupt, or queuing one or more occurrences of the interrupt for possible later delivery when a user-defined handler is attached to that interrupt.

30 NOTE 2 It is a bounded error to call Task_Identification.Current_Task (see C.7.1) from an interrupt handler.

31 NOTE 3 The rule that an exception propagated from an interrupt handler has no effect is modeled after the rule about exceptions propagated out of task bodies.

C.3.1 Protected Procedure Handlers

Paragraphs 1 through 6 were moved to Annex J, "Obsolescent Features".

Static Semantics

6.1/3 For a parameterless protected procedure, the following language-defined representation aspects may be specified:

6.2/3 Interrupt_Handler

The type of aspect Interrupt_Handler is Boolean. If directly specified, the aspect_definition shall be a static expression. This aspect is never inherited; if not directly specified, the aspect is False.

6.3/3 Attach_Handler

The aspect Attach_Handler is an expression, which shall be of type Interrupts.Interrupt_Id. This aspect is never inherited.

Legality Rules

7/3 If either the Attach_Handler or Interrupt_Handler aspect are specified for a protected procedure, the corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration and shall not be declared within a generic body. In addition to the places where Legality Rules normally apply (see 12.3), this rule also applies in the private part of an instance of a generic unit.

8/3 *This paragraph was deleted.*

Dynamic Semantics

If the Interrupt_Handler aspect of a protected procedure is True, then the procedure may be attached dynamically, as a handler, to interrupts (see C.3.2). Such procedures are allowed to be attached to multiple interrupts. 9/3

The expression specified for the Attach_Handler aspect of a protected procedure *P* is evaluated as part of the creation of the protected object that contains *P*. The value of the expression identifies an interrupt. As part of the initialization of that object, *P* (the *handler* procedure) is attached to the identified interrupt. A check is made that the corresponding interrupt is not reserved. Program_Error is raised if the check fails, and the existing treatment for the interrupt is not affected. 10/3

If the Ceiling_Locking policy (see D.3) is in effect, then upon the initialization of a protected object that contains a protected procedure for which either the Attach_Handler aspect is specified or the Interrupt_Handler aspect is True, a check is made that the initial ceiling priority of the object is in the range of System.Interrupt_Priority. If the check fails, Program_Error is raised. 11/3

When a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the Interrupts package or if no user handler was previously attached to the interrupt, the default treatment is restored. If the Attach_Handler aspect was specified and the most recently attached handler for the same interrupt is the same as the one that was attached at the time the protected object was initialized, the previous handler is restored. 12/3

When a handler is attached to an interrupt, the interrupt is blocked (subject to the Implementation Permission in C.3) during the execution of every protected action on the protected object containing the handler. 13

If restriction No_Dynamic_Attachment is in effect, then a check is made that the interrupt identified by an Attach_Handler aspect does not appear in any previously elaborated Attach_Handler aspect; Program_Error is raised if this check fails. 13.1/5

Erroneous Execution

If the Ceiling_Locking policy (see D.3) is in effect and an interrupt is delivered to a handler, and the interrupt hardware priority is higher than the ceiling priority of the corresponding protected object, the execution of the program is erroneous. 14

If the handlers for a given interrupt attached via aspect Attach_Handler are not attached and detached in a stack-like (LIFO) order, program execution is erroneous. In particular, when a protected object is finalized, the execution is erroneous if any of the procedures of the protected object are attached to interrupts via aspect Attach_Handler and the most recently attached handler for the same interrupt is not the same as the one that was attached at the time the protected object was initialized. 14.1/3

Metrics

The following metric shall be documented by the implementation: 15

- The worst-case overhead for an interrupt handler that is a parameterless protected procedure, in clock cycles. This is the execution time not directly attributable to the handler procedure or the interrupted execution. It is estimated as C – (A+B), where A is how long it takes to complete a given sequence of instructions without any interrupt, B is how long it takes to complete a normal call to a given protected procedure, and C is how long it takes to complete the same sequence of instructions when it is interrupted by one execution of the same procedure called via an interrupt. 16/2

Implementation Permissions

17/3 When the aspects Attach_Handler or Interrupt_Handler are specified for a protected procedure, the implementation is allowed to impose implementation-defined restrictions on the corresponding protected_type_declaration and protected_body.

18 An implementation may use a different mechanism for invoking a protected procedure in response to a hardware interrupt than is used for a call to that protected procedure from a task.

19/3 Notwithstanding what this subclause says elsewhere, the Attach_Handler and Interrupt_Handler aspects are allowed to be used for other, implementation defined, forms of interrupt handlers.

Implementation Advice

20 Whenever possible, the implementation should allow interrupt handlers to be called directly by the hardware.

21 Whenever practical, the implementation should detect violations of any implementation-defined restrictions before run time.

22/5 NOTE 1 The Attach_Handler aspect can provide static attachment of handlers to interrupts if the implementation supports preelaboration of protected objects. (See C.4.)

23/5 NOTE 2 For a protected object that has a (protected) procedure attached to an interrupt, the correct ceiling priority is at least as high as the highest processor priority at which that interrupt will ever be delivered.

24 NOTE 3 Protected procedures can also be attached dynamically to interrupts via operations declared in the predefined package Interrupts.

25 NOTE 4 An example of a possible implementation-defined restriction is disallowing the use of the standard storage pools within the body of a protected procedure that is an interrupt handler.

C.3.2 The Package Interrupts

Static Semantics

1 The following language-defined packages exist:

2/5
```
with System;
with System.Multiprocessors;
package Ada.Interrupts
   with Nonblocking, Global => in out synchronized is
   type Interrupt_Id is implementation-defined;
   type Parameterless_Handler is
      access protected procedure
      with Nonblocking => False;
```

3/1 *This paragraph was deleted.*

4
```
   function Is_Reserved (Interrupt : Interrupt_Id)
      return Boolean;
```

5
```
   function Is_Attached (Interrupt : Interrupt_Id)
      return Boolean;
```

6
```
   function Current_Handler (Interrupt : Interrupt_Id)
      return Parameterless_Handler;
```

7
```
   procedure Attach_Handler
      (New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_Id);
```

8
```
   procedure Exchange_Handler
      (Old_Handler : out Parameterless_Handler;
       New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_Id);
```

```
   procedure Detach_Handler
      (Interrupt : in Interrupt_Id);

   function Reference (Interrupt : Interrupt_Id)
      return System.Address;

   function Get_CPU (Interrupt : Interrupt_Id)
      return System.Multiprocessors.CPU_Range;

private
   ... -- not specified by the language
end Ada.Interrupts;

package Ada.Interrupts.Names
   with Nonblocking, Global => null is
   implementation-defined : constant Interrupt_Id :=
      implementation-defined;
      . . .
   implementation-defined : constant Interrupt_Id :=
      implementation-defined;
end Ada.Interrupts.Names;
```

Dynamic Semantics

The Interrupt_Id type is an implementation-defined discrete type used to identify interrupts. 13

The Is_Reserved function returns True if and only if the specified interrupt is reserved. 14

The Is_Attached function returns True if and only if a user-specified interrupt handler is attached to the interrupt. 15

The Current_Handler function returns a value that represents the attached handler of the interrupt. If no user-defined handler is attached to the interrupt, Current_Handler returns **null**. 16/1

The Attach_Handler procedure attaches the specified handler to the interrupt, overriding any existing treatment (including a user handler) in effect for that interrupt. If New_Handler is **null**, the default treatment is restored. If New_Handler designates a protected procedure for which the aspect Interrupt_Handler is False, Program_Error is raised. In this case, the operation does not modify the existing interrupt treatment. 17/3

The Exchange_Handler procedure operates in the same manner as Attach_Handler with the addition that the value returned in Old_Handler designates the previous treatment for the specified interrupt. If the previous treatment is not a user-defined handler, **null** is returned. 18/1

The Detach_Handler procedure restores the default treatment for the specified interrupt. 19

For all operations defined in this package that take a parameter of type Interrupt_Id, with the exception of Is_Reserved and Reference, a check is made that the specified interrupt is not reserved. Program_Error is raised if this check fails. 20

If, by using the Attach_Handler, Detach_Handler, or Exchange_Handler procedures, an attempt is made to detach a handler that was attached statically (using the aspect Attach_Handler), the handler is not detached and Program_Error is raised. 21/3

The Reference function returns a value of type System.Address that can be used to attach a task entry via an address clause (see J.7.1) to the interrupt specified by Interrupt. This function raises Program_Error if attaching task entries to interrupts (or to this particular interrupt) is not supported. 22/2

The function Get_CPU returns the processor on which the handler for Interrupt is executed. If the handler can execute on more than one processor the value System.Multiprocessors.Not_A_Specific_CPU is returned. 22.1/3

Implementation Requirements

23 At no time during attachment or exchange of handlers shall the current handler of the corresponding interrupt be undefined.

Documentation Requirements

24/5 The implementation shall document, when the Ceiling_Locking policy (see D.3) is in effect, the default ceiling priority assigned to a protected object that contains a protected procedure that specifies either the Attach_Handler or Interrupt_Handler aspects, but does not specify the Interrupt_Priority aspect. This default can be different for different interrupts.

Implementation Advice

25 If implementation-defined forms of interrupt handler procedures are supported, such as protected procedures with parameters, then for each such form of a handler, a type analogous to Parameterless_-Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts.

26 NOTE The package Interrupts.Names contains implementation-defined names (and constant values) for the interrupts that are supported by the implementation.

Examples

27 *Example of interrupt handlers:*

28/5
```
Device_Priority : constant
  array (Ada.Interrupts.Interrupt_Id range 1..5) of
    System.Interrupt_Priority := ( ... );
protected type Device_Interface
  (Int_Id : Ada.Interrupts.Interrupt_Id)
     with Interrupt_Priority => Device_Priority(Int_Id) is
  procedure Handler
     with Attach_Handler => Int_Id;
  ...
end Device_Interface;
  ...
Device_1_Driver : Device_Interface(1);
  ...
Device_5_Driver : Device_Interface(5);
  ...
```

C.4 Preelaboration Requirements

1/5 This subclause specifies additional implementation and documentation requirements for the Preelaborate aspect (see 10.2.1).

Implementation Requirements

2 The implementation shall not incur any run-time overhead for the elaboration checks of subprograms and protected_bodies declared in preelaborated library units.

3 The implementation shall not execute any memory write operations after load time for the elaboration of constant objects declared immediately within the declarative region of a preelaborated library package, so long as the subtype and initial expression (or default initial expressions if initialized by default) of the object_declaration satisfy the following restrictions. The meaning of *load time* is implementation defined.

4 • Any subtype_mark denotes a statically constrained subtype, with statically constrained subcomponents, if any;

- no subtype_mark denotes a controlled type, a private type, a private extension, a generic formal private type, a generic formal derived type, or a descendant of such a type; 4.1/2
- any constraint is a static constraint; 5
- any allocator is for an access-to-constant type; 6
- any uses of predefined operators appear only within static expressions; 7
- any primaries that are names, other than attribute_references for the Access or Address attributes, appear only within static expressions; 8
- any name that is not part of a static expression is an expanded name or direct_name that statically names some entity; 9/5
- any discrete_choice of an array_aggregate is static; 10
- no language-defined check associated with the elaboration of the object_declaration can fail. 11

Documentation Requirements

The implementation shall document any circumstances under which the elaboration of a preelaborated package causes code to be executed at run time. 12

The implementation shall document whether the method used for initialization of preelaborated variables allows a partition to be restarted without reloading. 13

Implementation Advice

It is recommended that preelaborated packages be implemented in such a way that there should be little or no code executed at run time for the elaboration of entities not already covered by the Implementation Requirements. 14

C.5 Aspect Discard_Names

Specifying the aspect Discard_Names can be used to request a reduction in storage used for the names of entities with runtime name text. 1/4

Static Semantics

An entity with *runtime name text* is a nonderived enumeration first subtype, a tagged first subtype, or an exception. 1.1/4

For an entity with runtime name text, the following language-defined representation aspect may be specified: 1.2/4

Discard_Names 1.3/4

The type of aspect Discard_Names is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

Syntax

The form of a pragma Discard_Names is as follows: 2

pragma Discard_Names[([On =>] local_name)]; 3

A pragma Discard_Names is allowed only immediately within a declarative_part, immediately within a package_specification, or as a configuration pragma. 4

Legality Rules

5/4 The local_name (if present) shall denote an entity with runtime name text. The pragma specifies that the aspect Discard_Names for the type or exception has the value True. Without a local_name, the pragma specifies that all entities with runtime name text declared after the pragma, within the same declarative region have the value True for aspect Discard_Names. Alternatively, the pragma can be used as a configuration pragma. If the configuration pragma Discard_Names applies to a compilation unit, all entities with runtime name text declared in the compilation unit have the value True for the aspect Discard_Names.

Static Semantics

6 If a local_name is given, then a pragma Discard_Names is a representation pragma.

7/6 If the aspect Discard_Names is True for an enumeration type, then the semantics of the default implementation of the Put_Image attribute is implementation defined for that type; the semantics of Image, Wide_Image, and Wide_Wide_Image are still defined in terms of Put_Image. Similarly, the semantics of the Wide_Wide_Value attribute is implementation defined for such a type; the semantics of Value, and Wide_Value are still defined in terms of Wide_Wide_Value. In addition, the semantics of Text_IO.Enumeration_IO are implementation defined. If the aspect Discard_Names is True for a tagged type, then the semantics of the Tags.Wide_Wide_Expanded_Name function are implementation defined for that type; the semantics of Tags.Expanded_Name and Tags.Wide_Expanded_Name are still defined in terms of Tags.Wide_Wide_Expanded_Name. If the aspect Discard_Names is True for an exception, then the semantics of the Exceptions.Wide_Wide_Exception_Name function are implementation defined for that exception; the semantics of Exceptions.Exception_Name and Exceptions.Wide_Exception_Name are still defined in terms of Exceptions.Wide_Wide_Exception_Name.

Implementation Advice

8/4 If the aspect Discard_Names is True for an entity, then the implementation should reduce the amount of storage used for storing names associated with that entity.

C.6 Shared Variable Control

1/3 This subclause defines representation aspects that control the use of shared variables.

Paragraphs 2 through 6 were moved to Annex J, "Obsolescent Features".

Static Semantics

6.1/5 For an object_declaration, a component_declaration, a full_type_declaration, or a formal_complete_type_declaration, the following representation aspects may be specified:

6.2/3 Atomic The type of aspect Atomic is Boolean.

6.3/3 Independent The type of aspect Independent is Boolean.

6.4/3 Volatile The type of aspect Volatile is Boolean.

6.5/5 Full_Access_Only
The type of aspect Full_Access_Only is Boolean.

6.6/5 For a full_type_declaration of an array type, an object_declaration for an object of an anonymous array type, or the formal_complete_type_declaration of a formal array type, the following representation aspects may be specified:

Atomic_Components 6.7/3
The type of aspect Atomic_Components is Boolean.

Volatile_Components 6.8/3
The type of aspect Volatile_Components is Boolean.

For a full_type_declaration of a composite type, an object_declaration for an object of an anonymous composite type, or the formal_complete_type_declaration of a formal composite type, the following representation aspect may be specified: 6.9/5

Independent_Components 6.10/3
The type of aspect Independent_Components is Boolean.

If any of these aspects are directly specified, the aspect_definition shall be a static expression. If not specified for a type (including by inheritance), the Atomic, Atomic_Components, and Full_Access_Only aspects are False. If any of these aspects are specified True for a type, then the corresponding aspect is True for all objects of the type. If the Atomic aspect is specified True, then the aspects Volatile, Independent, and Volatile_Component (if defined) are True; if the Atomic_Components aspect is specified True, then the aspects Volatile, Volatile_Components, and Independent_Components are True. If the Volatile aspect is specified True, then the Volatile_Components aspect (if defined) is True, and vice versa. When not determined by one of the other aspects, or for an object by its type, the Volatile, Volatile_Components, Independent, and Independent_Components aspects are False. 6.11/5

An *atomic* type is one for which the aspect Atomic is True. An *atomic* object (including a component) is one for which the aspect Atomic is True, or a component of an array for which the aspect Atomic_Components is True for the associated type, or any object of an atomic type, other than objects obtained by evaluating a slice. 7/3

A *volatile* type is one for which the aspect Volatile is True. A *volatile* object (including a component) is one for which the aspect Volatile is True, or a component of an array for which the aspect Volatile_-Components is True for the associated type, or any object of a volatile type. In addition, every atomic type or object is also defined to be volatile. Finally, if an object is volatile, then so are all of its subcomponents (the same does not apply to atomic). 8/3

When True, the aspects Independent and Independent_Components *specify as independently addressable* the named object or component(s), or in the case of a type, all objects or components of that type. All atomic objects and aliased objects are considered to be specified as independently addressable. 8.1/4

The Full_Access_Only aspect shall not be specified unless the associated type or object is volatile (or atomic). A *full access* type is any atomic type, or a volatile type for which the aspect Full_Access_Only is True. A *full access* object (including a component) is any atomic object, or a volatile object for which the aspect Full_Access_Only is True for the object or its type. A Full_Access_Only aspect is illegal if any subcomponent of the object or type is a full access object or is of a generic formal type. 8.2/5

Paragraph 9 was moved to Annex J, "Obsolescent Features".

Legality Rules

If aspect Independent_Components is specified for a full_type_declaration, the declaration shall be that of an array or record type. 9.1/3

It is illegal to specify either of the aspects Atomic or Atomic_Components to have the value True for an object or type if the implementation cannot support the indivisible and independent reads and updates required by the aspect (see below). 10/4

11/4 It is illegal to specify the Size attribute of an atomic object, the Component_Size attribute for an array type with atomic components, or the layout attributes of an atomic component, in a way that prevents the implementation from performing the required indivisible and independent reads and updates.

12/6 If an atomic object is passed as a parameter, then the formal parameter shall either have an atomic type or allow pass by copy. If an atomic object is used as an actual for a generic formal object of mode **in out**, then the type of the generic formal object shall be atomic. If the prefix of an attribute_reference for an Access attribute denotes an atomic object (including a component), then the designated type of the resulting access type shall be atomic. An extension of a nonatomic tagged type shall not be atomic. Corresponding rules apply to volatile objects and to full access objects.

12.1/5 If a nonatomic subcomponent of a full access object is passed as an actual parameter in a call then the formal parameter shall allow pass by copy (and, at run time, the parameter shall be passed by copy). A nonatomic subcomponent of a full access object shall not be used as an actual for a generic formal of mode **in out**. The prefix of an attribute_reference for an Access attribute shall not denote a nonatomic subcomponent of a full access object.

12.2/6 In evaluating a type conversion, a new object is created (as opposed to a new view of the operand object) if one but not both of the target type and the operand object is atomic. A corresponding rule applies in the case of volatile. In the case of an array conversion, corresponding rules apply if the Atomic_Components or Volatile_Components aspects of the array types do not match.

12.3/6 If the Atomic, Atomic_Components, Volatile, Volatile_Components, Independent, Independent_Components, or Full_Access_Only aspect is True for a generic formal type, then that aspect shall be True for the actual type. If an atomic type is used as an actual for a generic formal derived type, then the ancestor of the formal type shall be atomic. A corresponding rule applies to volatile types and similarly to full access types.

12.4/6 If a type with volatile components is used as an actual for a generic formal array type, then the components of the formal type shall be volatile. Furthermore, if the actual type has atomic components and the formal array type has aliased components, then the components of the formal array type shall also be atomic. A corresponding rule applies when the actual type has volatile full access components.

13/3 If an aspect Volatile, Volatile_Components, Atomic, or Atomic_Components is directly specified to have the value True for a stand-alone constant object, then the aspect Import shall also be specified as True for it.

13.1/3 It is illegal to specify the aspect Independent or Independent_Components as True for a component, object or type if the implementation cannot provide the independent addressability required by the aspect (see 9.10).

13.2/3 It is illegal to specify a representation aspect for a component, object or type for which the aspect Independent or Independent_Components is True, in a way that prevents the implementation from providing the independent addressability required by the aspect.

Paragraph 14 was moved to Annex J, "Obsolescent Features".

Dynamic Semantics

15 For an atomic object (including an atomic component) all reads and updates of the object as a whole are indivisible.

All tasks of the program (on all processors) that read or update volatile variables see the same order of updates to the variables. A use of an atomic variable or other mechanism may be necessary to avoid erroneous execution and to ensure that access to nonatomic volatile variables is sequential (see 9.10). 16/3

Two actions are sequential (see 9.10) if each is the read or update of the same atomic object. 17

If a type is atomic or volatile and it is not a by-copy type, then the type is defined to be a by-reference type. If any subcomponent of a type is atomic or volatile, then the type is defined to be a by-reference type. 18

If an actual parameter is atomic or volatile, and the corresponding formal parameter is not, then the parameter is passed by copy. 19

All reads of or writes to any nonatomic subcomponent of a full access object are performed by reading and/or writing all of the nearest enclosing full access object. 19.1/5

Implementation Requirements

The external effect of a program (see 1.1.3) is defined to include each read and update of a volatile or atomic object. The implementation shall not generate any memory reads or updates of atomic or volatile objects other than those specified by the program. However, there may be target-dependent cases where reading or writing a volatile but nonatomic object (typically a component) necessarily involves reading and/or writing neighboring storage, and that neighboring storage can overlap a volatile object. 20/5

This paragraph was deleted. 21/4

Implementation Permissions

Within the body of an instance of a generic unit that has a formal type *T* that is not atomic and an actual type that is atomic, if an object *O* of type *T* is declared and explicitly specified as atomic, the implementation may introduce an additional copy on passing *O* to a subprogram with a parameter of type *T* that is normally passed by reference. A corresponding permission applies to volatile parameter passing. 21.1/5

Implementation Advice

A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others, except in the case of a volatile but nonatomic subcomponent of an atomic object. 22/5

A load or store of an atomic object should, where possible, be implemented by a single load or store instruction. 23/2

NOTE 1 An imported volatile or atomic constant behaves as a constant (i.e. read-only) with respect to other parts of the Ada program, but can still be modified by an “external source”. 24

NOTE 2 Specifying the Pack aspect cannot override the effect of specifying an Atomic or Atomic_Components aspect. 25/4

NOTE 3 When mapping an Ada object to a memory-mapped hardware register, the Ada object can be declared atomic to ensure that the compiler will read and write exactly the bits of the register as specified in the source code and no others. 26/5

C.6.1 The Package System.Atomic_Operations

1/5 The language-defined package System.Atomic_Operations is the parent of a set of child units that provide facilities for manipulating objects of atomic types and for supporting lock-free synchronization. The subprograms of this subsystem are Intrinsic subprograms (see 6.3.1) in order to provide convenient access to machine operations that can provide these capabilities if they are available in the target environment.

Static Semantics

2/5 The library package System.Atomic_Operations has the following declaration:

3/5
```
package System.Atomic_Operations
   with Pure, Nonblocking is
end System.Atomic_Operations;
```

4/5 System.Atomic_Operations serves as the parent of other language-defined library units that manipulate atomic objects; its declaration is empty.

5/5 A call to a subprogram is said to be *lock-free* if the subprogram is guaranteed to return from the call while keeping the processor of the logical thread of control busy for the duration of the call.

6/5 In each child package, a function Is_Lock_Free(...) is provided to check whether the operations of the child package can all be provided lock-free for a given object. Is_Lock_Free returns True if operations defined in the child package are lock-free when applied to the object denoted by Item, and Is_Lock_Free returns False otherwise.

C.6.2 The Package System.Atomic_Operations.Exchange

1/5 The language-defined generic package System.Atomic_Operations.Exchange provides the following operations:

2/5 - To atomically compare the value of two atomic objects, and update the first atomic object with a desired value if both objects were found to be equal, or otherwise update the second object with the value of the first object.

3/5 - To atomically update the value of an atomic object, and then return the value that the atomic object had just prior to the update.

Static Semantics

4/5 The generic library package System.Atomic_Operations.Exchange has the following declaration:

5/5
```
generic
   type Atomic_Type is private with Atomic;
package System.Atomic_Operations.Exchange
   with Pure, Nonblocking is
```

6/5
```
   function Atomic_Exchange (Item  : aliased in out Atomic_Type;
                             Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;
```

7/5
```
   function Atomic_Compare_And_Exchange
     (Item    : aliased in out Atomic_Type;
      Prior   : aliased in out Atomic_Type;
      Desired : Atomic_Type) return Boolean
      with Convention => Intrinsic;
```

8/5
```
   function Is_Lock_Free (Item : aliased Atomic_Type) return Boolean
      with Convention => Intrinsic;
```

9/5
```
end System.Atomic_Operations.Exchange;
```

Atomic_Exchange atomically assigns the value of Value to Item, and returns the previous value of Item. 10/5

Atomic_Compare_And_Exchange first evaluates the value of Prior. Atomic_Compare_And_Exchange then performs the following steps as part of a single indivisible operation: 11/5

- evaluates the value of Item; 12/5
- compares the value of Item with the value of Prior; 13/5
- if equal, assigns Item the value of Desired; 14/5
- otherwise, makes no change to the value of Item. 15/5

After these steps, if the value of Item and Prior did not match, Prior is assigned the original value of Item, and the function returns False. Otherwise, Prior is unaffected and the function returns True. 16/5

Examples

Example of a spin lock using Atomic_Exchange: 17/5

```
type Atomic_Boolean is new Boolean with Atomic;                                    18/5
package Exchange is new
   Atomic_Operations.Exchange (Atomic_Type => Atomic_Boolean);
Lock : aliased Atomic_Boolean := False;                                            19/5
...                                                                                20/5
begin -- Some critical section, trying to get the lock:                            21/5
   -- Obtain the lock                                                             22/5
   while Exchange.Atomic_Exchange (Item => Lock, Value => True) loop
      null;
   end loop;
   ... -- Do stuff                                                                 23/5
   Lock := False; -- Release the lock                                              24/5
end;
```

C.6.3 The Package System.Atomic_Operations.Test_and_Set

The language-defined package System.Atomic_Operations.Test_And_Set provides an operation to atomically set and clear an atomic flag object. 1/5

Static Semantics

The library package System.Atomic_Operations.Test_And_Set has the following declaration: 2/5

```
package System.Atomic_Operations.Test_And_Set                                      3/5
   with Pure, Nonblocking is
   type Test_And_Set_Flag is mod implementation-defined                            4/5
      with Atomic, Default_Value => 0, Size => implementation-defined;
   function Atomic_Test_And_Set                                                    5/5
     (Item : aliased in out Test_And_Set_Flag) return Boolean
      with Convention => Intrinsic;
   procedure Atomic_Clear                                                          6/5
     (Item : aliased in out Test_And_Set_Flag)
      with Convention => Intrinsic;
   function Is_Lock_Free                                                           7/5
     (Item : aliased Test_And_Set_Flag) return Boolean
      with Convention => Intrinsic;
end System.Atomic_Operations.Test_And_Set;                                         8/5
```

Test_And_Set_Flag represents the state of an atomic flag object. An atomic flag object can either be considered to be set or cleared. 9/5

10/5 Atomic_Test_And_Set performs an atomic test-and-set operation on Item. Item is set to some implementation-defined nonzero value. The function returns True if the previous contents were nonzero, and otherwise returns False.

11/5 Atomic_Clear performs an atomic clear operation on Item. After the operation, Item contains 0. This call should be used in conjunction with Atomic_Test_And_Set.

C.6.4 The Package System.Atomic_Operations.Integer_Arithmetic

1/5 The language-defined generic package System.Atomic_Operations.Integer_Arithmetic provides operations to perform arithmetic atomically on objects of integer types.

Static Semantics

2/5 The generic library package System.Atomic_Operations.Integer_Arithmetic has the following declaration:

3/5
```
generic
   type Atomic_Type is range <> with Atomic;
package System.Atomic_Operations.Integer_Arithmetic
   with Pure, Nonblocking is
```

4/5
```
   procedure Atomic_Add (Item  : aliased in out Atomic_Type;
                         Value : Atomic_Type)
      with Convention => Intrinsic;
```

5/5
```
   procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;
                              Value : Atomic_Type)
      with Convention => Intrinsic;
```

6/5
```
   function Atomic_Fetch_And_Add
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;
```

7/5
```
   function Atomic_Fetch_And_Subtract
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;
```

8/5
```
   function Is_Lock_Free (Item : aliased Atomic_Type) return Boolean
      with Convention => Intrinsic;
```

9/5
```
end System.Atomic_Operations.Integer_Arithmetic;
```

10/5 The operations of this package are defined as follows:

11/5
```
procedure Atomic_Add (Item  : aliased in out Atomic_Type;
                      Value : Atomic_Type)
   with Convention => Intrinsic;
```

12/5 Atomically performs: `Item := Item + Value;`

13/5
```
procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;
                           Value : Atomic_Type)
   with Convention => Intrinsic;
```

14/5 Atomically performs: `Item := Item - Value;`

15/5
```
function Atomic_Fetch_And_Add
  (Item  : aliased in out Atomic_Type;
   Value : Atomic_Type) return Atomic_Type
   with Convention => Intrinsic;
```

16/5 Atomically performs: `Tmp := Item; Item := Item + Value; return Tmp;`

```
function Atomic_Fetch_And_Subtract
  (Item  : aliased in out Atomic_Type;
   Value : Atomic_Type) return Atomic_Type
   with Convention => Intrinsic;
```

17/5

Atomically performs: `Tmp := Item; Item := Item - Value; return Tmp;` 18/5

Implementation Permissions

An implementation may allow `Item` to be modified if the operation defined in an instance of `System.Atomic_Operations.Integer_Arithmetic` fails a language-defined check. Additionally, an implementation may check that predicates are satisfied after the atomic operation. 19/6

Usage

The subtype used to instantiate `System.Atomic_Operations.Integer_Arithmetic` should not have predicates, as those probably will not be checked atomically, potentially causing incorrect check failures if another task has modified the value between the operation and the check. For the best performance, the subtype should be a base subtype so that no range check need be performed. 20/6

C.6.5 The Package System.Atomic_Operations.Modular_Arithmetic

The language-defined generic package System.Atomic_Operations.Modular_Arithmetic provides operations to perform arithmetic atomically on objects of modular types. 1/5

Static Semantics

The generic library package System.Atomic_Operations.Modular_Arithmetic has the following declaration: 2/5

```
generic                                                                          3/5
   type Atomic_Type is mod <> with Atomic;
package System.Atomic_Operations.Modular_Arithmetic
   with Pure, Nonblocking is

   procedure Atomic_Add (Item  : aliased in out Atomic_Type;                     4/5
                         Value : Atomic_Type)
      with Convention => Intrinsic;

   procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;                5/5
                              Value : Atomic_Type)
      with Convention => Intrinsic;

   function Atomic_Fetch_And_Add                                                 6/5
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;

   function Atomic_Fetch_And_Subtract                                            7/5
     (Item  : aliased in out Atomic_Type;
      Value : Atomic_Type) return Atomic_Type
      with Convention => Intrinsic;

   function Is_Lock_Free (Item : aliased Atomic_Type) return Boolean             8/5
      with Convention => Intrinsic;

end System.Atomic_Operations.Modular_Arithmetic;                                 9/5
```

The operations of this package are defined as follows: 10/5

```
procedure Atomic_Add (Item  : aliased in out Atomic_Type;                        11/5
                      Value : Atomic_Type)
   with Convention => Intrinsic;
```

Atomically performs: `Item := Item + Value;` 12/5

13/5
```
procedure Atomic_Subtract (Item  : aliased in out Atomic_Type;
                           Value : Atomic_Type)
   with Convention => Intrinsic;
```

14/5 Atomically performs: `Item := Item - Value;`

15/5
```
function Atomic_Fetch_And_Add
  (Item  : aliased in out Atomic_Type;
   Value : Atomic_Type) return Atomic_Type
   with Convention => Intrinsic;
```

16/5 Atomically performs: `Tmp := Item; Item := Item + Value; return Tmp;`

17/5
```
function Atomic_Fetch_And_Subtract
  (Item  : aliased in out Atomic_Type;
   Value : Atomic_Type) return Atomic_Type
   with Convention => Intrinsic;
```

18/5 Atomically performs: `Tmp := Item; Item := Item - Value; return Tmp;`

Implementation Permissions

19/6 An implementation may allow `Item` to be modified if the operation defined in an instance of `System.Atomic_Operations.Modular_Arithmetic` fails a language-defined check. Additionally, an implementation may check that predicates are satisfied after the atomic operation.

Usage

20/6 The subtype used to instantiate `System.Atomic_Operations.Modular_Arithmetic` should not have predicates, as those probably will not be checked atomically, potentially causing incorrect check failures if another task has modified the value between the operation and the check. For the best performance, the subtype should be a base subtype so that no range check need be performed.

C.7 Task Information

This subclause describes operations and attributes that can be used to obtain the identity of a task. In addition, a package that associates user-defined information with a task is defined. Finally, a package that associates termination procedures with a task or set of tasks is defined. 1/3

C.7.1 The Package Task_Identification

Static Semantics

The following language-defined library package exists: 1

```
package Ada.Task_Identification                                                    2/5
   with Preelaborate, Nonblocking, Global => in out synchronized is
   type Task_Id is private
      with Preelaborable_Initialization;
   Null_Task_Id : constant Task_Id;
   function  "=" (Left, Right : Task_Id) return Boolean;

   function  Image                  (T : Task_Id) return String;                   3/5
   function  Current_Task     return Task_Id;
   function  Environment_Task return Task_Id;
   procedure Abort_Task             (T : in Task_Id)
      with Nonblocking => False;

   function  Is_Terminated          (T : Task_Id) return Boolean;                  4/3
   function  Is_Callable            (T : Task_Id) return Boolean;
   function  Activation_Is_Complete (T : Task_Id) return Boolean;
private
   ... -- not specified by the language
end Ada.Task_Identification;
```

Dynamic Semantics

A value of the type Task_Id identifies an existent task. The constant Null_Task_Id does not identify any task. Each object of the type Task_Id is default initialized to the value of Null_Task_Id. 5

The function "=" returns True if and only if Left and Right identify the same task or both have the value Null_Task_Id. 6

The function Image returns an implementation-defined string that identifies T. If T equals Null_Task_Id, Image returns an empty string. 7

The function Current_Task returns a value that identifies the calling task. 8

The function Environment_Task returns a value that identifies the environment task. 8.1/3

The effect of Abort_Task is the same as the abort_statement for the task identified by T. In addition, if T identifies the environment task, the entire partition is aborted, see E.1. 9/5

The functions Is_Terminated and Is_Callable return the value of the corresponding attribute of the task identified by T. 10

The function Activation_Is_Complete returns True if the task identified by T has completed its activation (whether successfully or not). It returns False otherwise. If T identifies the environment task, Activation_Is_Complete returns True after the elaboration of the library_items of the partition has completed. 10.1/3

For a prefix T that is of a task type (after any implicit dereference), the following attribute is defined: 11

12 T'Identity Yields a value of the type Task_Id that identifies the task denoted by T.

13 For a prefix E that denotes an entry_declaration, the following attribute is defined:

14/3 E'Caller Yields a value of the type Task_Id that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an accept_statement, or entry_body after the entry_barrier, corresponding to the entry_declaration denoted by E.

15/5 Program_Error is raised if a value of Null_Task_Id is passed as a parameter to Abort_Task, Activation_Is_Complete, Is_Terminated, and Is_Callable.

16/5 *This paragraph was deleted.*

Bounded (Run-Time) Errors

17/3 It is a bounded error to call the Current_Task function from an entry_body, interrupt handler, or finalization of a task attribute. Program_Error is raised, or an implementation-defined value of the type Task_Id is returned.

Erroneous Execution

18 If a value of Task_Id is passed as a parameter to any of the operations declared in this package (or any language-defined child of this package), and the corresponding task object no longer exists, the execution of the program is erroneous.

Documentation Requirements

19 The implementation shall document the effect of calling Current_Task from an entry body or interrupt handler.

20 NOTE 1 This package is intended for use in writing user-defined task scheduling packages and constructing server tasks. Current_Task can be used in conjunction with other operations requiring a task as an argument such as Set_Priority (see D.5).

21 NOTE 2 The function Current_Task and the attribute Caller can return a Task_Id value that identifies the environment task.

C.7.2 The Package Task_Attributes

Static Semantics

1 The following language-defined generic library package exists:

2/5
```
with Ada.Task_Identification; use Ada.Task_Identification;
generic
   type Attribute is private;
   Initial_Value : in Attribute;
package Ada.Task_Attributes
   with Nonblocking, Global => in out synchronized is
```

3
```
   type Attribute_Handle is access all Attribute;
```

4
```
   function Value(T : Task_Id := Current_Task)
     return Attribute;
```

5
```
   function Reference(T : Task_Id := Current_Task)
     return Attribute_Handle;
```

6
```
   procedure Set_Value(Val : in Attribute;
                       T : in Task_Id := Current_Task);
   procedure Reinitialize(T : in Task_Id := Current_Task);
```

7
```
end Ada.Task_Attributes;
```

Dynamic Semantics

When an instance of Task_Attributes is elaborated in a given active partition, an object of the actual type corresponding to the formal type Attribute is implicitly created for each task (of that partition) that exists and is not yet terminated. This object acts as a user-defined attribute of the task. A task created previously in the partition and not yet terminated has this attribute from that point on. Each task subsequently created in the partition will have this attribute when created. In all these cases, the initial value of the given attribute is Initial_Value. 8

The Value operation returns the value of the corresponding attribute of T. 9

The Reference operation returns an access value that designates the corresponding attribute of T. 10

The Set_Value operation performs any finalization on the old value of the attribute of T and assigns Val to that attribute (see 5.2 and 7.6). 11

The effect of the Reinitialize operation is the same as Set_Value where the Val parameter is replaced with Initial_Value. 12

For all the operations declared in this package, Tasking_Error is raised if the task identified by T is terminated. Program_Error is raised if the value of T is Null_Task_Id. 13

After a task has terminated, all of its attributes are finalized, unless they have been finalized earlier. When the master of an instantiation of Ada.Task_Attributes is finalized, the corresponding attribute of each task is finalized, unless it has been finalized earlier. 13.1/2

Bounded (Run-Time) Errors

If the package Ada.Task_Attributes is instantiated with a controlled type and the controlled type has user-defined Adjust or Finalize operations that in turn access task attributes by any of the above operations, then a call of Set_Value of the instantiated package constitutes a bounded error. The call may perform as expected or may result in forever blocking the calling task and subsequently some or all tasks of the partition. 13.2/1

Erroneous Execution

It is erroneous to dereference the access value returned by a given call on Reference after a subsequent call on Reinitialize for the same task attribute, or after the associated task terminates. 14

If a value of Task_Id is passed as a parameter to any of the operations declared in this package and the corresponding task object no longer exists, the execution of the program is erroneous. 15

An access to a task attribute via a value of type Attribute_Handle is erroneous if executed concurrently with another such access or a call of any of the operations declared in package Task_Attributes. An access to a task attribute is erroneous if executed concurrently with or after the finalization of the task attribute. 15.1/2

Implementation Requirements

For a given attribute of a given task, the implementation shall perform the operations declared in this package atomically with respect to any of these operations of the same attribute of the same task. The granularity of any locking mechanism necessary to achieve such atomicity is implementation defined. 16/1

After task attributes are finalized, the implementation shall reclaim any storage associated with the attributes. 17/2

Documentation Requirements

18 The implementation shall document the limit on the number of attributes per task, if any, and the limit on the total storage for attribute values per task, if such a limit exists.

19 In addition, if these limits can be configured, the implementation shall document how to configure them.

Metrics

20/2 The implementation shall document the following metrics: A task calling the following subprograms shall execute at a sufficiently high priority as to not be preempted during the measurement period. This period shall start just before issuing the call and end just after the call completes. If the attributes of task T are accessed by the measurement tests, no other task shall access attributes of that task during the measurement period. For all measurements described here, the Attribute type shall be a scalar type whose size is equal to the size of the predefined type Integer. For each measurement, two cases shall be documented: one where the accessed attributes are of the calling task (that is, the default value for the T parameter is used), and the other, where T identifies another, nonterminated, task.

21 The following calls (to subprograms in the Task_Attributes package) shall be measured:

22 • a call to Value, where the return value is Initial_Value;

23 • a call to Value, where the return value is not equal to Initial_Value;

24 • a call to Reference, where the return value designates a value equal to Initial_Value;

25 • a call to Reference, where the return value designates a value not equal to Initial_Value;

26/2 • a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is equal to Initial_Value;

27 • a call to Set_Value where the Val parameter is not equal to Initial_Value and the old attribute value is not equal to Initial_Value.

Implementation Permissions

28/5 An implementation can avoid actually creating the object corresponding to a task attribute until its value is set to something other than that of Initial_Value, or until Reference is called for the task attribute. Similarly, when the value of the attribute is to be reinitialized to that of Initial_Value, the object may instead be finalized and its storage reclaimed, to be recreated when needed later. While the object does not exist, the function Value may simply return Initial_Value, rather than implicitly creating the object.

29 An implementation is allowed to place restrictions on the maximum number of attributes a task may have, the maximum size of each attribute, and the total storage size allocated for all the attributes of a task.

Implementation Advice

30/5 Some implementations are targeted to domains in which memory use at run time has to be completely deterministic. For such implementations, it is recommended that the storage for task attributes will be pre-allocated statically and not from the heap. This can be accomplished by either placing restrictions on the number and the size of the attributes of a task, or by using the pre-allocated storage for the first N attribute objects, and the heap for the others. In the latter case, N should be documented.

30.1/2 Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination.

31/5 NOTE 1 An attribute always exists (after instantiation), and has the initial value. An implementation can avoid using memory to store the attribute value until the first operation that changes the attribute value. The same holds true after Reinitialize.

NOTE 2 The result of the Reference function is always safe to use in the task body whose attribute is being accessed. However, when the result is being used by another task, the programmer will want to make sure that the task whose attribute is being accessed is not yet terminated. Failing to do so can make the program execution erroneous. 32/5

C.7.3 The Package Task_Termination

Static Semantics

The following language-defined library package exists: 1/2

```
with Ada.Task_Identification;                                                    2/5
with Ada.Exceptions;
package Ada.Task_Termination
   with Preelaborate, Nonblocking, Global => in out synchronized is

   type Cause_Of_Termination is (Normal, Abnormal, Unhandled_Exception);         3/2

   type Termination_Handler is access protected procedure                        4/2
     (Cause : in Cause_Of_Termination;
      T     : in Ada.Task_Identification.Task_Id;
      X     : in Ada.Exceptions.Exception_Occurrence);

   procedure Set_Dependents_Fallback_Handler                                     5/2
     (Handler: in Termination_Handler);
   function Current_Task_Fallback_Handler return Termination_Handler;

   procedure Set_Specific_Handler                                                6/2
     (T       : in Ada.Task_Identification.Task_Id;
      Handler : in Termination_Handler);
   function Specific_Handler (T : Ada.Task_Identification.Task_Id)
      return Termination_Handler;

end Ada.Task_Termination;                                                        7/2
```

Dynamic Semantics

The type Termination_Handler identifies a protected procedure to be executed by the implementation when a task terminates. Such a protected procedure is called a *handler*. In all cases T identifies the task that is terminating. If the task terminates due to completing the last statement of its body, or as a result of waiting on a terminate alternative, and the finalization of the task completes normally, then Cause is set to Normal and X is set to Null_Occurrence. If the task terminates because it is being aborted, then Cause is set to Abnormal; X is set to Null_Occurrence if the finalization of the task completes normally. If the task terminates because of an exception raised by the execution of its task_body, then Cause is set to Unhandled_Exception; X is set to the associated exception occurrence if the finalization of the task completes normally. Independent of how the task completes, if finalization of the task propagates an exception, then Cause is either Unhandled_Exception or Abnormal, and X is an exception occurrence that identifies the Program_Error exception. 8/3

Each task has two termination handlers, a *fall-back handler* and a *specific handler*. The specific handler applies only to the task itself, while the fall-back handler applies only to the dependent tasks of the task. A handler is said to be *set* if it is associated with a nonnull value of type Termination_Handler, and *cleared* otherwise. When a task is created, its specific handler and fall-back handler are cleared. 9/2

The procedure Set_Dependents_Fallback_Handler changes the fall-back handler for the calling task: if Handler is **null**, that fall-back handler is cleared; otherwise, it is set to be Handler.**all**. If a fall-back handler had previously been set it is replaced. 10/3

The function Current_Task_Fallback_Handler returns the fall-back handler that is currently set for the calling task, if one is set; otherwise, it returns **null**. 11/3

12/3 The procedure Set_Specific_Handler changes the specific handler for the task identified by T: if Handler is **null**, that specific handler is cleared; otherwise, it is set to be Handler.**all**. If a specific handler had previously been set it is replaced.

13/3 The function Specific_Handler returns the specific handler that is currently set for the task identified by T, if one is set; otherwise, it returns **null**.

14/2 As part of the finalization of a task_body, after performing the actions specified in 7.6 for finalization of a master, the specific handler for the task, if one is set, is executed. If the specific handler is cleared, a search for a fall-back handler proceeds by recursively following the master relationship for the task. If a task is found whose fall-back handler is set, that handler is executed; otherwise, no handler is executed.

15/2 For Set_Specific_Handler or Specific_Handler, Tasking_Error is raised if the task identified by T has already terminated. Program_Error is raised if the value of T is Ada.Task_Identification.Null_Task_Id.

16/2 An exception propagated from a handler that is invoked as part of the termination of a task has no effect.

Erroneous Execution

17/2 For a call of Set_Specific_Handler or Specific_Handler, if the task identified by T no longer exists, the execution of the program is erroneous.

Annex D
(normative)
Real-Time Systems

This Annex specifies additional characteristics of Ada implementations intended for real-time systems software. To conform to this Annex, an implementation shall also conform to Annex C, “Systems Programming”. 1/5

Metrics

The metrics are documentation requirements; an implementation shall document the values of the language-defined metrics for at least one configuration of hardware or an underlying system supported by the implementation, and shall document the details of that configuration. 2

The metrics do not necessarily yield a simple number. For some, a range is more suitable, for others a formula dependent on some parameter is appropriate, and for others, it may be more suitable to break the metric into several cases. Unless specified otherwise, the metrics in this annex are expressed in processor clock cycles. For metrics that require documentation of an upper bound, if there is no upper bound, the implementation shall report that the metric is unbounded. 3

NOTE 1 The specification of the metrics makes a distinction between upper bounds and simple execution times. Where something is just specified as “the execution time of” a piece of code, this leaves one the freedom to choose a nonpathological case. This kind of metric is of the form “there exists a program such that the value of the metric is V”. Conversely, the meaning of upper bounds is “there is no program such that the value of the metric is greater than V”. This kind of metric can only be partially tested, by finding the value of V for one or more test programs. 4

NOTE 2 The metrics do not cover the whole language; they are limited to features that are specified in Annex C, “Systems Programming” and in this Annex. The metrics are intended to provide guidance to potential users as to whether a particular implementation of such a feature is going to be adequate for a particular real-time application. As such, the metrics are aimed at known implementation choices that can result in significant performance differences. 5

NOTE 3 The purpose of the metrics is not necessarily to provide fine-grained quantitative results or to serve as a comparison between different implementations on the same or different platforms. Instead, their goal is rather qualitative; to define a standard set of approximate values that can be measured and used to estimate the general suitability of an implementation, or to evaluate the comparative utility of certain features of an implementation for a particular real-time application. 6

D.1 Task Priorities

This subclause specifies the priority model for real-time systems. In addition, the methods for specifying priorities are defined. 1/3

Paragraphs 2 through 6 were moved to Annex J, “Obsolescent Features”.

Static Semantics

For a task type (including the anonymous type of a single_task_declaration), protected type (including the anonymous type of a single_protected_declaration), or subprogram, the following language-defined representation aspects may be specified: 6.1/3

Priority The aspect Priority is an expression, which shall be of type Integer. 6.2/3

Interrupt_Priority 6.3/3

The aspect Interrupt_Priority is an expression, which shall be of type Integer.

Legality Rules

7/3 *This paragraph was deleted.*

8/3 If the Priority aspect is specified for a subprogram, the expression shall be static, and its value shall be in the range of System.Priority.

8.1/3 At most one of the Priority and Interrupt_Priority aspects may be specified for a given entity.

8.2/3 Neither of the Priority or Interrupt_Priority aspects shall be specified for a synchronized interface type.

8.3/6 An internal call shall not occur within the expression of a Priority or Interrupt_Priority aspect of a protected type.

Static Semantics

9 The following declarations exist in package System:

10
```
subtype Any_Priority is Integer range implementation-defined;
subtype Priority is Any_Priority
   range Any_Priority'First .. implementation-defined;
subtype Interrupt_Priority is Any_Priority
   range Priority'Last+1 .. Any_Priority'Last;
```

11
```
Default_Priority : constant Priority := (Priority'First + Priority'Last)/2;
```

12 The full range of priority values supported by an implementation is specified by the subtype Any_Priority. The subrange of priority values that are high enough to require the blocking of one or more interrupts is specified by the subtype Interrupt_Priority. The subrange of priority values below System.Interrupt_-Priority'First is specified by the subtype System.Priority.

13/3 *This paragraph was deleted.*

Dynamic Semantics

14/3 The Priority aspect has no effect if it is specified for a subprogram other than the main subprogram; the Priority value is not associated with any task.

15/5 A *task priority* is an integer value that indicates a degree of urgency and is the basis for resolving competing demands of tasks for resources. Unless otherwise specified, whenever tasks compete for processors or other implementation-defined resources, the resources are allocated to the task with the highest priority value. The *base priority* of a task is the priority with which it was created, or to which it was later set by Dynamic_Priorities.Set_Priority (see D.5). At all times, a task also has an *active priority*, which generally is its base priority unless it inherits a priority from other sources. *Priority inheritance* is the process by which the priority of a task or other entity (for example, a protected object; see D.3) is used in the evaluation of another task's active priority.

16/3 The effect of specifying a Priority or Interrupt_Priority aspect for a protected type or single_protected_declaration is discussed in D.3.

17/4 The expression specified for the Priority or Interrupt_Priority aspect of a task type is evaluated each time an object of the task type is created (see 9.1). For the Priority aspect, the value of the expression is converted to the subtype Priority; for the Interrupt_Priority aspect, this value is converted to the subtype Any_Priority. The priority value is then associated with the task object.

18/3 Likewise, the priority value is associated with the environment task if the aspect is specified for the main subprogram.

19/3 The initial value of a task's base priority is specified by default or by means of a Priority or Interrupt_Priority aspect. After a task is created, its base priority can be changed only by a call to

Dynamic_Priorities.Set_Priority (see D.5). The initial base priority of a task in the absence of an aspect is the base priority of the task that creates it at the time of creation (see 9.1). If the aspect Priority is not specified for the main subprogram, the initial base priority of the environment task is System.Default_Priority. The task's active priority is used when the task competes for processors. Similarly, the task's active priority is used to determine the task's position in any queue when Priority_Queuing is specified (see D.4).

At any time, the active priority of a task is the maximum of all the priorities the task is inheriting at that instant. For a task that is not held (see D.11), its base priority is a source of priority inheritance unless otherwise specified for a particular task dispatching policy. Other sources of priority inheritance are specified under the following conditions: 20/2

- During activation, a task being activated inherits the active priority that its activator (see 9.2) had at the time the activation was initiated. 21/1
- During rendezvous, the task accepting the entry call inherits the priority of the entry call (see 9.5.3 and D.4). 22/1
- While starting a protected action on a protected object when the FIFO_Spinning admission policy is in effect, a task inherits the ceiling priority of the protected object (see 9.5, D.3, and D.4.1). 22.1/5
- While a task executes a protected action on a protected object, the task inherits the ceiling priority of the protected object (see 9.5 and D.3). 23/5

In all of these cases, the priority ceases to be inherited as soon as the condition calling for the inheritance no longer exists. 24

Implementation Requirements

The range of System.Interrupt_Priority shall include at least one value. 25

The range of System.Priority shall include at least 30 values. 26

NOTE 1 The priority expression can include references to discriminants of the enclosing type. 27

NOTE 2 It is a consequence of the active priority rules that at the point when a task stops inheriting a priority from another source, its active priority is re-evaluated. This is in addition to other instances described in this Annex for such re-evaluation. 28

NOTE 3 An implementation can provide a nonstandard mode in which tasks inherit priorities under conditions other than those specified above. 29/5

D.2 Priority Scheduling

This subclause describes the rules that determine which task is selected for execution when more than one task is ready (see 9). 1/3

D.2.1 The Task Dispatching Model

1/2 The task dispatching model specifies task scheduling, based on conceptual priority-ordered ready queues.

Static Semantics

1.1/2 The following language-defined library package exists:

1.2/5
```
package Ada.Dispatching
  with Preelaborate, Nonblocking, Global => in out synchronized is
```

1.3/5
```
   procedure Yield
    with Nonblocking => False;
```

1.4/3
```
   Dispatching_Policy_Error : exception;
end Ada.Dispatching;
```

1.5/2 Dispatching serves as the parent of other language-defined library units concerned with task dispatching.

1.6/5 For a noninstance subprogram (including a generic formal subprogram), a generic subprogram, or an entry, the following language-defined aspect may be specified with an aspect_specification (see 13.1.1):

1.7/5 Yield The type of aspect Yield is Boolean.

1.8/5 If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False.

1.9/5 If a Yield aspect is specified True for a primitive subprogram *S* of a type *T*, then the aspect is inherited by the corresponding primitive subprogram of each descendant of *T*.

Legality Rules

1.10/5 If the Yield aspect is specified for a dispatching subprogram that inherits the aspect, the specified value shall be confirming.

1.11/5 If the Nonblocking aspect (see 9.5) of the associated callable entity is statically True, the Yield aspect shall not be specified as True. For a callable entity that is declared within a generic body, this rule is checked assuming that any nonstatic Nonblocking attributes in the expression of the Nonblocking aspect of the entity are statically True.

1.12/5 In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Dynamic Semantics

2/5 A task can become a *running task* only if it is ready (see Clause 9) and the execution resources required by that task are available. Processors are allocated to tasks based on each task's active priority.

3 It is implementation defined whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy.

4/5 *Task dispatching* is the process by which a logical thread of control associated with a ready task is selected for execution on a processor. This selection is done during the execution of such a logical thread of control, at certain points called *task dispatching points*. Such a logical thread of control reaches a task dispatching point whenever it becomes blocked, and when its associated task terminates. Other task dispatching points are defined throughout this Annex for specific policies. Below we talk in terms of tasks, but in the context of a parallel construct, a single task can be represented by multiple logical threads of control, each of which can appear separately on a ready queue.

Task dispatching policies are specified in terms of conceptual *ready queues* and task states. A ready queue is an ordered list of ready tasks. The first position in a queue is called the *head of the queue*, and the last position is called the *tail of the queue*. A task is *ready* if it is in a ready queue, or if it is running. Each processor has one ready queue for each priority value. At any instant, each ready queue of a processor contains exactly the set of tasks of that priority that are ready for execution on that processor, but are not running on any processor; that is, those tasks that are ready, are not running on any processor, and can be executed using that processor and other available resources. A task can be on the ready queues of more than one processor. 5/2

Each processor also has one *running task*, which is the task currently being executed by that processor. Whenever a task running on a processor reaches a task dispatching point it goes back to one or more ready queues; a task (possibly the same task) is then selected to run on that processor. The task selected is the one at the head of the highest priority nonempty ready queue; this task is then removed from all ready queues to which it belongs. 6/2

A call of Yield and a delay_statement are task dispatching points for all language-defined policies. 7/5

If the Yield aspect has the value True, then a call to procedure Yield is included within the body of the associated callable entity, and invoked immediately prior to returning from the body if and only if no other task dispatching points were encountered during the execution of the body. 8/5

Implementation Permissions

An implementation is allowed to define additional resources as execution resources, and to define the corresponding allocation policies for them. Such resources may have an implementation-defined effect on task dispatching. 9/2

An implementation may place implementation-defined restrictions on tasks whose active priority is in the Interrupt_Priority range. 10

Unless otherwise specified for a task dispatching policy, an implementation may add additional points at which task dispatching may occur, in an implementation-defined manner. 10.1/5

NOTE 1 Clause 9 specifies under which circumstances a task becomes ready. The ready state is affected by the rules for task activation and termination, delay statements, and entry calls. When a task is not ready, it is said to be blocked. 11

NOTE 2 An example of a possible implementation-defined execution resource is a page of physical memory, which must be loaded with a particular page of virtual memory before a task can continue execution. 12/5

NOTE 3 The ready queues are purely conceptual; there is no requirement that such lists physically exist in an implementation. 13

NOTE 4 While a task is running, it is not on any ready queue. Any time the task that is running on a processor is added to a ready queue, a new running task is selected for that processor. 14

NOTE 5 In a multiprocessor system, a task can be on the ready queues of more than one processor. At the extreme, if several processors share the same set of ready tasks, the contents of their ready queues is identical, and so they can be viewed as sharing one ready queue, and can be implemented that way. Thus, the dispatching model covers multiprocessors where dispatching is implemented using a single ready queue, as well as those with separate dispatching domains. 15

NOTE 6 The priority of a task is determined by rules specified in this subclause, and under D.1, “Task Priorities”, D.3, “Priority Ceiling Locking”, and D.5, “Dynamic Priorities”. 16

NOTE 7 The setting of a task's base priority as a result of a call to Set_Priority does not always take effect immediately when Set_Priority is called. The effect of setting the task's base priority is deferred while the affected task performs a protected action. 17/2

D.2.2 Task Dispatching Pragmas

0.1/3 This subclause allows a single task dispatching policy to be defined for all priorities, or the range of priorities to be split into subranges that are assigned individual dispatching policies.

Syntax

1 The form of a pragma Task_Dispatching_Policy is as follows:

2 **pragma** Task_Dispatching_Policy(*policy_*identifier);

2.1/2 The form of a pragma Priority_Specific_Dispatching is as follows:

2.2/2 **pragma** Priority_Specific_Dispatching (
*policy_*identifier, *first_priority_*expression, *last_priority_*expression);

Name Resolution Rules

2.3/2 The expected type for *first_priority_*expression and *last_priority_*expression is Integer.

Legality Rules

3/2 The *policy_*identifier used in a pragma Task_Dispatching_Policy shall be the name of a task dispatching policy.

3.1/2 The *policy_*identifier used in a pragma Priority_Specific_Dispatching shall be the name of a task dispatching policy.

3.2/2 Both *first_priority_*expression and *last_priority_*expression shall be static expressions in the range of System.Any_Priority; *last_priority_*expression shall have a value greater than or equal to *first_priority_*-expression.

Static Semantics

3.3/2 Pragma Task_Dispatching_Policy specifies the single task dispatching policy.

3.4/2 Pragma Priority_Specific_Dispatching specifies the task dispatching policy for the specified range of priorities. Tasks with base priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma have their active priorities determined according to the specified dispatching policy. Tasks with active priorities within the range of priorities specified in a Priority_Specific_Dispatching pragma are dispatched according to the specified dispatching policy.

3.5/3 If a partition contains one or more Priority_Specific_Dispatching pragmas, the dispatching policy for priorities not covered by any Priority_Specific_Dispatching pragmas is FIFO_Within_Priorities.

Post-Compilation Rules

4/2 A Task_Dispatching_Policy pragma is a configuration pragma. A Priority_Specific_Dispatching pragma is a configuration pragma.

4.1/2 The priority ranges specified in more than one Priority_Specific_Dispatching pragma within the same partition shall not be overlapping.

4.2/2 If a partition contains one or more Priority_Specific_Dispatching pragmas it shall not contain a Task_Dispatching_Policy pragma.

5/2 *This paragraph was deleted.*

Dynamic Semantics

A *task dispatching policy* specifies the details of task dispatching that are not covered by the basic task dispatching model. These rules govern when tasks are inserted into and deleted from the ready queues. A single task dispatching policy is specified by a Task_Dispatching_Policy pragma. Pragma Priority_Specific_Dispatching assigns distinct dispatching policies to subranges of System.Any_Priority. 6/2

If neither pragma applies to any of the program units comprising a partition, the task dispatching policy for that partition is unspecified. 6.1/2

If a partition contains one or more Priority_Specific_Dispatching pragmas, a task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. 6.2/3

A task that has its base priority changed may move from one dispatching policy to another. It is immediately subject to the new dispatching policy. 6.3/2

Paragraphs 7 through 13 were moved to D.2.3.

Implementation Requirements

An implementation shall allow, for a single partition, both the locking policy (see D.3) to be specified as Ceiling_Locking and also one or more Priority_Specific_Dispatching pragmas to be given. 13.1/2

Documentation Requirements

Paragraphs 14 through 16 were moved to D.2.3.

Implementation Permissions

Implementations are allowed to define other task dispatching policies, but are not required to support specifying more than one task dispatching policy per partition. 17/5

An implementation is not required to support pragma Priority_Specific_Dispatching if it is infeasible to support it in the target environment. 18/5

Paragraphs 19 through 21 were deleted.

D.2.3 Preemptive Dispatching

This subclause defines a preemptive task dispatching policy. 1/3

Static Semantics

The *policy_*identifier FIFO_Within_Priorities is a task dispatching policy. 2/2

Dynamic Semantics

When FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows: 3/2

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority. 4/2
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority, except in the case where the active priority is lowered due to the loss of inherited priority, in which case the task is added at the head of the ready queue for its new active priority. 5/2

6/2 • When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority.

7/2 • When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority.

8/2 Each of the events specified above is a task dispatching point (see D.2.1).

9/2 A task dispatching point occurs for the currently running task of a processor whenever there is a nonempty ready queue for that processor with a higher priority than the priority of the running task. The currently running task is said to be *preempted* and it is added at the head of the ready queue for its active priority.

Implementation Requirements

10/2 An implementation shall allow, for a single partition, both the task dispatching policy to be specified as FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

Documentation Requirements

11/2 *Priority inversion* is the duration for which a task remains at the head of the highest priority nonempty ready queue while the processor executes a lower priority task. The implementation shall document:

12/2 • The maximum priority inversion a user task can experience due to activity of the implementation (on behalf of lower priority tasks), and

13/2 • whether execution of a task can be preempted by the implementation processing of delay expirations for lower priority tasks, and if so, for how long.

14/2 NOTE 1 If the active priority of a running task is lowered due to loss of inherited priority (as it is on completion of a protected operation) and there is a ready task of the same active priority that is not running, the running task continues to run (provided that there is no higher priority task).

15/2 NOTE 2 Setting the base priority of a ready task causes the task to move to the tail of the queue for its active priority, regardless of whether the active priority of the task actually changes.

D.2.4 Non-Preemptive Dispatching

1/3 This subclause defines a non-preemptive task dispatching policy.

Static Semantics

2/2 The *policy_*identifier Non_Preemptive_FIFO_Within_Priorities is a task dispatching policy.

2.1/3 The following language-defined library package exists:

2.2/5
```
package Ada.Dispatching.Non_Preemptive
  with Preelaborate, Nonblocking, Global => in out synchronized is
  procedure Yield_To_Higher;
  procedure Yield_To_Same_Or_Higher renames Yield;
end Ada.Dispatching.Non_Preemptive;
```

2.3/3 A call of Yield_To_Higher is a task dispatching point for this policy. If the task at the head of the highest priority ready queue has a higher active priority than the calling task, then the calling task is preempted.

Legality Rules

3/2 Non_Preemptive_FIFO_Within_Priorities shall not be specified as the *policy_*identifier of pragma Priority_Specific_Dispatching (see D.2.2).

Dynamic Semantics

When Non_Preemptive_FIFO_Within_Priorities is in effect, modifications to the ready queues occur only as follows: 4/2

- When a blocked task becomes ready, it is added at the tail of the ready queue for its active priority. 5/2
- When the active priority of a ready task that is not running changes, or the setting of its base priority takes effect, the task is removed from the ready queue for its old active priority and is added at the tail of the ready queue for its new active priority. 6/2
- When the setting of the base priority of a running task takes effect, the task is added to the tail of the ready queue for its active priority. 7/2
- When a task executes a delay_statement that does not result in blocking, it is added to the tail of the ready queue for its active priority. 8/2

For this policy, blocking or termination of a task, a delay_statement, a call to Yield_To_Higher, and a call to Yield_To_Same_Or_Higher or Yield are the only task dispatching points (see D.2.1). 9/3

Implementation Requirements

An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Non_Preemptive_FIFO_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking. 10/2

Implementation Permissions

Since implementations are allowed to round up to System.Priority'Last all ceiling priorities in subrange System.Priority'Range (see D.3), an implementation may allow a task of a partition using the Non_Premptive_FIFO_Within_Priorities policy to execute within a protected object without raising its active priority provided the associated protected unit does not contain any subprograms with aspects Interrupt_Handler or Attach_Handler specified, nor does the unit have aspect Interrupt_Priority specified. When the locking policy (see D.3) is Ceiling_Locking, an implementation taking advantage of this permission shall ensure that a call to Yield_to_Higher that occurs within a protected action uses the ceiling priority of the protected object (rather than the active priority of the task) when determining whether to preempt the task. 11/6

D.2.5 Round Robin Dispatching

This subclause defines the task dispatching policy Round_Robin_Within_Priorities and the package Round_Robin. 1/3

Static Semantics

The *policy*_identifier Round_Robin_Within_Priorities is a task dispatching policy. 2/2

3/2 The following language-defined library package exists:

4/5
```
with System;
with Ada.Real_Time;
package Ada.Dispatching.Round_Robin
  with Nonblocking, Global => in out synchronized is
  Default_Quantum : constant Ada.Real_Time.Time_Span :=
              implementation-defined;
  procedure Set_Quantum (Pri     : in System.Priority;
                         Quantum : in Ada.Real_Time.Time_Span);
  procedure Set_Quantum (Low, High : in System.Priority;
                         Quantum   : in Ada.Real_Time.Time_Span);
  function Actual_Quantum (Pri : System.Priority)
             return Ada.Real_Time.Time_Span;
  function Is_Round_Robin (Pri : System.Priority) return Boolean;
end Ada.Dispatching.Round_Robin;
```

5/2 When task dispatching policy Round_Robin_Within_Priorities is the single policy in effect for a partition, each task with priority in the range of System.Interrupt_Priority is dispatched according to policy FIFO_Within_Priorities.

Dynamic Semantics

6/2 The procedures Set_Quantum set the required Quantum value for a single priority level Pri or a range of priority levels Low .. High. If no quantum is set for a Round Robin priority level, Default_Quantum is used.

7/2 The function Actual_Quantum returns the actual quantum used by the implementation for the priority level Pri.

8/3 The function Is_Round_Robin returns True if priority Pri is covered by task dispatching policy Round_Robin_Within_Priorities; otherwise, it returns False.

9/2 A call of Actual_Quantum or Set_Quantum raises exception Dispatching.Dispatching_Policy_Error if a predefined policy other than Round_Robin_Within_Priorities applies to the specified priority or any of the priorities in the specified range.

10/2 For Round_Robin_Within_Priorities, the dispatching rules for FIFO_Within_Priorities apply with the following additional rules:

11/2 - When a task is added or moved to the tail of the ready queue for its base priority, it has an execution time budget equal to the quantum for that priority level. This will also occur when a blocked task becomes executable again.

12/2 - When a task is preempted (by a higher priority task) and is added to the head of the ready queue for its priority level, it retains its remaining budget.

13/2 - While a task is executing, its budget is decreased by the amount of execution time it uses. The accuracy of this accounting is the same as that for execution time clocks (see D.14).

14/2 - When a task has exhausted its budget and is without an inherited priority (and is not executing within a protected operation), it is moved to the tail of the ready queue for its priority level. This is a task dispatching point.

Implementation Requirements

15/2 An implementation shall allow, for a single partition, both the task dispatching policy to be specified as Round_Robin_Within_Priorities and also the locking policy (see D.3) to be specified as Ceiling_Locking.

Documentation Requirements

16/2 An implementation shall document the quantum values supported.

An implementation shall document the accuracy with which it detects the exhaustion of the budget of a task. 17/2

NOTE 1 Due to implementation constraints, the quantum value returned by Actual_Quantum can differ from that set with Set_Quantum. 18/5

NOTE 2 A task that executes continuously with an inherited priority will not be subject to round robin dispatching. 19/2

D.2.6 Earliest Deadline First Dispatching

The deadline of a task is an indication of the urgency of the task; it represents a point on an ideal physical time line. The deadline can affect how resources are allocated to the task. 1/5

This subclause presents Dispatching.EDF, a package for representing the deadline of a task and a dispatching policy that defines Earliest Deadline First (EDF) dispatching. The Relative_Deadline aspect is provided to assign an initial deadline to a task. A configuration pragma Generate_Deadlines is provided to specify that a task's deadline is recomputed whenever it is made ready. 2/5

Paragraphs 3 through 6 were moved to Annex J, "Obsolescent Features".

Static Semantics

The *policy_*identifier EDF_Within_Priorities is a task dispatching policy. 7/5

The following language-defined library package exists: 8/2

```
with Ada.Real_Time;                                                    9/5
with Ada.Task_Identification;
package Ada.Dispatching.EDF
  with Nonblocking, Global => in out synchronized is
  subtype Deadline is Ada.Real_Time.Time;
  subtype Relative_Deadline is Ada.Real_Time.Time_Span;
  Default_Deadline : constant Deadline :=
             Ada.Real_Time.Time_Last;
  Default_Relative_Deadline : constant Relative_Deadline :=
             Ada.Real_Time.Time_Span_Last;
  procedure Set_Deadline
     (D : in Deadline;
      T : in Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task);
  function Get_Deadline
     (T : Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task) return Deadline;
  procedure Set_Relative_Deadline
     (D : in Relative_Deadline;
      T : in Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task);
  function Get_Relative_Deadline
     (T : Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task)
      return Relative_Deadline;
  procedure Delay_Until_And_Set_Deadline
     (Delay_Until_Time : in Ada.Real_Time.Time;
      Deadline_Offset : in Ada.Real_Time.Time_Span)
     with Nonblocking => False;
  function Get_Last_Release_Time
     (T : Ada.Task_Identification.Task_Id :=
      Ada.Task_Identification.Current_Task)
      return Ada.Real_Time.Time;
end Ada.Dispatching.EDF;
```

9.1/5 For a subprogram, a task type (including the anonymous type of a single_task_declaration), or a protected type (including the anonymous type of a single_protected_declaration), the following language-defined representation aspect may be specified:

9.2/3 Relative_Deadline

The aspect Relative_Deadline is an expression, which shall be of type Real_Time.Time_Span.

9.3/5 The form of pragma Generate_Deadlines is as follows:

9.4/5 **pragma** Generate_Deadlines;

9.5/5 The Generate_Deadlines pragma is a configuration pragma.

Legality Rules

9.6/5 The Relative_Deadline aspect shall not be specified on a task or protected interface type. If the Relative_Deadline aspect is specified for a subprogram, the aspect_definition shall be a static expression.

9.7/6 An internal call shall not occur within the expression of a Relative_Deadline aspect of a protected type.

Post-Compilation Rules

10/5 If the EDF_Within_Priorities policy is specified for a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

11/5 If the EDF_Within_Priorities policy appears in a Priority_Specific_Dispatching pragma (see D.2.2) in a partition, then the Ceiling_Locking policy (see D.3) shall also be specified for the partition.

Dynamic Semantics

12/3 The Relative_Deadline aspect has no effect if it is specified for a subprogram other than the main subprogram.

12.1/5 If pragma Generate_Deadlines is in effect, the deadline of a task is recomputed each time it becomes ready. The new deadline is the value of Real_Time.Clock at the time the task is added to a ready queue plus the value returned by Get_Relative_Deadline.

13/6 The initial absolute deadline for a task with a specified Relative_Deadline is the result of adding the results of evaluating both a call to Real_Time.Clock and the expression specified as the Relative_Deadline aspect, where both evaluations (and the addition) are performed between task creation and the start of its activation. If the aspect Relative_Deadline is not specified, then the initial absolute deadline of a task is the value of Default_Deadline (Ada.Real_Time.Time_Last). The environment task is also given an initial deadline by this rule, using the value of the Relative_Deadline aspect of the main subprogram (if any).]}

13.1/5 The effect of specifying a Relative_Deadline aspect for a protected type or single_protected_declaration is discussed in D.3.

14/5 A task has both an *active* and a *base* absolute deadline. These are the same except when the task is inheriting a relative deadline during activation or a rendezvous (see below) or within a protected action (see D.3). The procedure Set_Deadline changes the (base) absolute deadline of the task to D. The function Get_Deadline returns the (base) absolute deadline of the task.

14.1/5 The procedure Set_Relative_Deadline changes the relative deadline of the task to D. The function Get_Relative_Deadline returns the relative deadline of the task.

The function Get_Last_Release_Time returns the time, as provided by Real_Time.Clock, when the task was last made ready (that is, was added to a ready queue). 14.2/5

The procedure Delay_Until_And_Set_Deadline delays the calling task until time Delay_Until_Time. When the task becomes ready again it will have deadline `Delay_Until_Time + Deadline_Offset`. 15/5

On a system with a single processor, the setting of the deadline of a task to the new value occurs immediately at the first point that is outside the execution of a protected action. If the task is currently on a ready queue it is removed and re-entered onto the ready queue determined by the rules defined below. 16/5

When EDF_Within_Priorities is specified for a priority, the ready queue for that priority is ordered by deadline. The task at the head of a queue is the one with the earliest deadline. 17/5

A task dispatching point occurs for the currently running task *T* to which policy EDF_Within_Priorities applies: 18/5

- when a change to the base (absolute) deadline of *T* occurs; 19/5
- *This paragraph was deleted.* 20/5
- there is a nonempty ready queue for that processor with a higher priority than the active priority of the running task; 21/5
- there is a ready task with the same priority as *T* but with an earlier absolute deadline. 21.1/5

In these cases, the currently running task is said to be preempted and is returned to the ready queue for its active priority, at a position determined by its active (absolute) deadline. 22/5

Paragraphs 23 through 27 were deleted.

When the setting of the base priority of a ready task takes effect and the new priority is specified as EDF_Within_Priorities, the task is added to the ready queue, at a position determined by its active deadline. 28/5

For all the operations defined in Dispatching.EDF, Tasking_Error is raised if the task identified by T has terminated. Program_Error is raised if the value of T is Null_Task_Id. 29/2

If two tasks with priority designated as EDF_Within_Priorities rendezvous then the deadline for the execution of the accept statement is the earlier of the deadlines of the two tasks. 29.1/5

During activation, a task being activated inherits the deadline that its activator (see 9.2) had at the time the activation was initiated. 29.2/5

Paragraph 30 was deleted.

Erroneous Execution

If a value of Task_Id is passed as a parameter to any of the subprograms of this package and the corresponding task object no longer exists, the execution of the program is erroneous. 31/2

Documentation Requirements

On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the deadline of a task to be delayed later than what is specified for a single processor. 32/2

NOTE If two distinct priorities are specified to have policy EDF_Within_Priorities, then tasks from the higher priority always run before tasks of the lower priority, regardless of deadlines. 33/5

This paragraph was deleted. 34/5

D.3 Priority Ceiling Locking

1/3 This subclause specifies the interactions between priority task scheduling and protected object ceilings. This interaction is based on the concept of the *ceiling priority* of a protected object.

Syntax

2 The form of a pragma Locking_Policy is as follows:

3 **pragma** Locking_Policy(*policy*_identifier);

Legality Rules

4 The *policy*_identifier shall either be Ceiling_Locking or an implementation-defined identifier.

Post-Compilation Rules

5 A Locking_Policy pragma is a configuration pragma.

Dynamic Semantics

6/2 A locking policy specifies the details of protected object locking. All protected objects have a priority. The locking policy specifies the meaning of the priority of a protected object, and the relationships between these priorities and task priorities. In addition, the policy specifies the state of a task when it executes a protected action, and how its active priority is affected by the locking. The *locking policy* is specified by a Locking_Policy pragma. For implementation-defined locking policies, the meaning of the priority of a protected object is implementation defined. If no Locking_Policy pragma applies to any of the program units comprising a partition, the locking policy for that partition, as well as the meaning of the priority of a protected object, are implementation defined.

6.1/3 The expression specified for the Priority or Interrupt_Priority aspect (see D.1) is evaluated as part of the creation of the corresponding protected object and converted to the subtype System.Any_Priority or System.Interrupt_Priority, respectively. The value of the expression is the initial priority of the corresponding protected object. If no Priority or Interrupt_Priority aspect is specified for a protected object, the initial priority is specified by the locking policy.

7 There is one predefined locking policy, Ceiling_Locking; this policy is defined as follows:

8/3 - Every protected object has a *ceiling priority*, which is determined by either a Priority or Interrupt_Priority aspect as defined in D.1, or by assignment to the Priority attribute as described in D.5.2. The ceiling priority of a protected object (or ceiling, for short) is an upper bound on the active priority a task can have when it calls protected operations of that protected object.

9/2 - The initial ceiling priority of a protected object is equal to the initial priority for that object.

10/4 - If an Interrupt_Handler or Attach_Handler aspect (see C.3.1) is specified for a protected subprogram of a protected type that does not have either the Priority or Interrupt_Priority aspect specified, the initial priority of protected objects of that type is implementation defined, but in the range of the subtype System.Interrupt_Priority.

11/3 - If neither aspect Priority nor Interrupt_Priority is specified for a protected type, and no protected subprogram of the type has aspect Interrupt_Handler or Attach_Handler specified, then the initial priority of the corresponding protected object is System.Priority'Last.

12 - While a task executes a protected action, it inherits the ceiling priority of the corresponding protected object.

- When a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails. 13

If the task dispatching policy specified for the ceiling priority of a protected object is EDF_Within_Priorities, the following additional rules apply: 13.1/5

- Every protected object has a *relative deadline*, which is determined by a Relative_Deadline aspect as defined in D.2.6, or by assignment to the Relative_Deadline attribute as described in D.5.2. The relative deadline of a protected object represents a lower bound on the relative deadline a task may have when it calls a protected operation of that protected object. 13.2/5
- If aspect Relative_Deadline is not specified for a protected type then the initial relative deadline of the corresponding protected object is Ada.Real_Time.Time_Span_Zero. 13.3/5
- While a task executes a protected action on a protected object *P*, it inherits the relative deadline of *P*. In this case, let *DF* be 'now' ('now' is obtained via a call on Ada.Real_Time.Clock at the start of the action) plus the deadline floor of *P*. If the active deadline of the task is later than *DF*, its active deadline is reduced to *DF*; the active deadline is unchanged otherwise. 13.4/5
- When a task calls a protected operation, a check is made that its active deadline minus its last release time is not less than the relative deadline of the corresponding protected object; Program_Error is raised if this check fails. 13.5/5

Bounded (Run-Time) Errors

Following any change of priority, it is a bounded error for the active priority of any task with a call queued on an entry of a protected object to be higher than the ceiling priority of the protected object. In this case one of the following applies: 13.6/5

- at any time prior to executing the entry body, Program_Error is raised in the calling task; 13.7/5
- when the entry is open, the entry body is executed at the ceiling priority of the protected object; 13.8/5
- when the entry is open, the entry body is executed at the ceiling priority of the protected object and then Program_Error is raised in the calling task; or 13.9/5
- when the entry is open, the entry body is executed at the ceiling priority of the protected object that was in effect when the entry call was queued. 13.10/5

Implementation Permissions

The implementation is allowed to round all ceilings in a certain subrange of System.Priority or System.Interrupt_Priority up to the top of that subrange, uniformly. 14

Implementations are allowed to define other locking policies, but are not required to support specifying more than one locking policy per partition. 15/5

Since implementations are allowed to place restrictions on code that runs at an interrupt-level active priority (see C.3.1 and D.2.1), the implementation may implement a language feature in terms of a protected object with an implementation-defined ceiling, but the ceiling shall be no less than Priority'Last. 16

Implementation Advice

The implementation should use names that end with "_Locking" for implementation-defined locking policies. 17

NOTE 1 While a task executes in a protected action, it can be preempted only by tasks whose active priorities are higher than the ceiling priority of the protected object. 18

NOTE 2 If a protected object has a ceiling priority in the range of Interrupt_Priority, certain interrupts are blocked while protected actions of that object execute. In the extreme, if the ceiling is Interrupt_Priority'Last, all blockable interrupts are blocked during that time. 19

20/5 NOTE 3 As described in C.3.1, whenever an interrupt is handled by one of the protected procedures of a protected object, a check is made that its ceiling priority is in the Interrupt_Priority range.

21/5 NOTE 4 When specifying the ceiling of a protected object, a correct value is one that is at least as high as the highest active priority at which tasks can be executing when they call protected operations of that object. In determining this value the following factors, which can affect active priority, are relevant: the effect of Set_Priority, nested protected operations, entry calls, task activation, and other implementation-defined factors.

22 NOTE 5 Attaching a protected procedure whose ceiling is below the interrupt hardware priority to an interrupt causes the execution of the program to be erroneous (see C.3.1).

23 NOTE 6 On a single processor implementation, the ceiling priority rules guarantee that there is no possibility of deadlock involving only protected subprograms (excluding the case where a protected operation calls another protected operation on the same protected object).

D.4 Entry Queuing Policies

1/5 This subclause specifies a mechanism for a user to choose an entry *queuing policy*. It also defines three such policies. Other policies are implementation defined.

Syntax

2 The form of a pragma Queuing_Policy is as follows:

3 **pragma** Queuing_Policy(*policy_*identifier);

Legality Rules

4/5 The *policy_*identifier shall be either FIFO_Queuing, Ordered_FIFO_Queuing, Priority_Queuing or an implementation-defined identifier.

Post-Compilation Rules

5 A Queuing_Policy pragma is a configuration pragma.

Dynamic Semantics

6 A *queuing policy* governs the order in which tasks are queued for entry service, and the order in which different entry queues are considered for service. The queuing policy is specified by a Queuing_Policy pragma.

7/5 Three queuing policies, FIFO_Queuing, Ordered_FIFO_Queuing, and Priority_Queuing, are language defined. If no Queuing_Policy pragma applies to any of the program units comprising the partition, the queuing policy for that partition is FIFO_Queuing. The rules for the FIFO_Queuing policy are specified in 9.5.3 and 9.7.1.

7.1/5 The Ordered_FIFO_Queuing policy is defined as follows:

7.2/5 - Calls are selected on a given entry queue in order of arrival.

7.3/5 - When more than one condition of an entry_barrier of a protected object becomes True, and more than one of the respective queues is nonempty, the call that arrived first is selected.

7.4/5 - If the expiration time of two or more open delay_alternatives is the same and no other accept_alternatives are open, the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed.

7.5/5 - When more than one alternative of a selective_accept is open and has queued calls, the alternative whose queue has the call that arrived first is selected.

The Priority_Queuing policy is defined as follows: 8

- The calls to an entry (including a member of an entry family) are queued in an order consistent with the priorities of the calls. The *priority of an entry call* is initialized from the active priority of the calling task at the time the call is made, but can change later. Within the same priority, the order is consistent with the calling (or requeuing, or priority setting) time (that is, a FIFO order). 9
- After a call is first queued, changes to the active priority of a task do not affect the priority of the call, unless the base priority of the task is set while the task is blocked on an entry call. 10/1
- When the base priority of a task is set (see D.5), if the task is blocked on an entry call, and the call is queued, the priority of the call is updated to the new active priority of the calling task. This causes the call to be removed from and then reinserted in the queue at the new active priority. 11
- When more than one condition of an entry_barrier of a protected object becomes True, and more than one of the respective queues is nonempty, the call with the highest priority is selected. If more than one such call has the same priority, the call that is queued on the entry whose declaration is first in textual order in the protected_definition is selected. For members of the same entry family, the one with the lower family index is selected. 12
- If the expiration time of two or more open delay_alternatives is the same and no other accept_alternatives are open, the sequence_of_statements of the delay_alternative that is first in textual order in the selective_accept is executed. 13
- When more than one alternative of a selective_accept is open and has queued calls, an alternative whose queue has the highest-priority call at its head is selected. If two or more open alternatives have equal-priority queued calls, then a call on the entry in the accept_alternative that is first in textual order in the selective_accept is selected. 14

Implementation Permissions

Implementations are allowed to define other queuing policies, but are not required to support specifying more than one queuing policy per partition. 15/5

Implementations are allowed to defer the reordering of entry queues following a change of base priority of a task blocked on the entry call if it is not practical to reorder the queue immediately. 15.1/2

Implementation Advice

The implementation should use names that end with "_Queuing" for implementation-defined queuing policies. 16

Static Semantics

For a task type (including the anonymous type of a single_task_declaration), protected type (including the anonymous type of a single_protected_declaration), or an entry_declaration, the following language-defined representation aspect may be specified: 17/5

Max_Entry_Queue_Length 18/5

The type of aspect Max_Entry_Queue_Length is Integer.

If directly specified, the aspect_definition shall be a static expression no less than -1. If not specified, the aspect has value -1 (representing no additional restriction on queue length). 19/5

Legality Rules

If the Max_Entry_Queue_Length aspect for a type has a nonnegative value, the Max_Entry_Queue_Length aspect for every individual entry of that type shall not be greater than the value of the aspect for the type. The Max_Entry_Queue_Length aspect of a type is nonoverridable (see 13.1.1). 20/5

Post-Compilation Rules

21/5 If a restriction Max_Entry_Queue_Length applies to a partition, any value specified for the Max_Entry_Queue_Length aspect specified for the declaration of a type or entry in the partition shall not be greater than the value of the restriction.

Dynamic Semantics

22/5 If a nonconfirming value is specified for Max_Entry_Queue_Length for a type, and an entry call or requeue would cause the queue for any entry of the type to become longer than the specified value, then Program_Error is raised at the point of the call or requeue.

23/5 If a nonconfirming value is specified for Max_Entry_Queue_Length for an entry, and an entry call or requeue would cause the queue for an entry to become longer than the specified value, then Program_Error is raised at the point of the call or requeue.

D.4.1 Admission Policies

1/5 This subclause specifies a mechanism for a user to choose an admission policy. It also defines one such policy. Other policies are implementation defined.

Syntax

2/5 The form of a pragma Admission_Policy is as follows:

3/5 **pragma** Admission_Policy (*policy*_identifier);

Legality Rules

4/5 The *policy*_identifier shall be either FIFO_Spinning or an implementation-defined identifier.

Post-Compilation Rules

5/5 An Admission_Policy pragma is a configuration pragma.

Dynamic Semantics

6/5 An admission policy governs the order in which competing tasks are evaluated for acquiring the execution resource associated with a protected object. The admission policy is specified by an Admission_Policy pragma.

7/5 One admission policy, FIFO_Spinning, is language defined. If FIFO_Spinning is in effect, and starting a protected action on a protected object involves busy-waiting, then calls are selected for acquiring the execution resource of the protected object in the order in which the busy-wait was initiated; otherwise the FIFO_Spinning policy has no effect. If no Admission_Policy pragma applies to any of the program units in the partition, the admission policy for that partition is implementation defined.

Implementation Permissions

8/5 Implementations are allowed to define other admission policies, but are not required to support specifying more than one admission policy per partition.

D.5 Dynamic Priorities

1/3 This subclause describes how the priority of an entity can be modified or queried at run time.

D.5.1 Dynamic Priorities for Tasks

This subclause describes how the base priority of a task can be modified or queried at run time. 1/3

Static Semantics

The following language-defined library package exists: 2

```
with System;                                                                    3/5
with Ada.Task_Identification; -- See C.7.1
package Ada.Dynamic_Priorities
    with Preelaborate, Nonblocking, Global => in out synchronized is

    procedure Set_Priority(Priority : in System.Any_Priority;                   4
                           T : in Ada.Task_Identification.Task_Id :=
                           Ada.Task_Identification.Current_Task);

    function Get_Priority (T : Ada.Task_Identification.Task_Id :=               5
                           Ada.Task_Identification.Current_Task)
                           return System.Any_Priority;

end Ada.Dynamic_Priorities;                                                     6
```

Dynamic Semantics

The procedure Set_Priority sets the base priority of the specified task to the specified Priority value. Set_Priority has no effect if the task is terminated. 7

The function Get_Priority returns T's current base priority. Tasking_Error is raised if the task is terminated. 8

Program_Error is raised by Set_Priority and Get_Priority if T is equal to Null_Task_Id. 9

On a system with a single processor, the setting of the base priority of a task *T* to the new value occurs immediately at the first point when *T* is outside the execution of a protected action. 10/2

Paragraph 11 was deleted.

Erroneous Execution

If any subprogram in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous. 12

Documentation Requirements

On a multiprocessor, the implementation shall document any conditions that cause the completion of the setting of the priority of a task to be delayed later than what is specified for a single processor. 12.1/2

Metrics

The implementation shall document the following metric: 13

- The execution time of a call to Set_Priority, for the nonpreempting case, in processor clock cycles. This is measured for a call that modifies the priority of a ready task that is not running (which cannot be the calling one), where the new base priority of the affected task is lower than the active priority of the calling task, and the affected task is not on any entry queue and is not executing a protected operation. 14

  NOTE 1 Setting a task's base priority affects task dispatching. First, it can change the task's active priority. Second, under the FIFO_Within_Priorities policy it always causes the task to move to the tail of the ready queue corresponding to its active priority, even if the new base priority is unchanged. 15/2

  NOTE 2 Under the priority queuing policy, setting a task's base priority has an effect on a queued entry call if the task is blocked waiting for the call. That is, setting the base priority of a task causes the priority of a queued entry call from that 16

task to be updated and the call to be removed and then reinserted in the entry queue at the new priority (see D.4), unless the call originated from the triggering_statement of an asynchronous_select.

17 NOTE 3 The effect of two or more Set_Priority calls executed in parallel on the same task is defined as executing these calls in some serial order.

18/5 NOTE 4 The rule for when Tasking_Error is raised for Set_Priority or Get_Priority is different from the rule for when Tasking_Error is raised on an entry call (see 9.5.3). In particular, the priority of a completed or an abnormal task can be queried so long as the task is not yet terminated, and the priority of a task can be set in any task state (including for terminated tasks).

19/5 NOTE 5 Changing the priorities of a set of tasks can be performed by a series of calls to Set_Priority for each task separately. This can be done reliably within a protected operation that has high enough ceiling priority to guarantee that the operation completes without being preempted by any of the affected tasks.

D.5.2 Dynamic Priorities for Protected Objects

1/3 This subclause specifies how the priority of a protected object can be modified or queried at run time.

Static Semantics

2/5 The following attributes are defined for a prefix P that denotes a protected object:

3/2 P'Priority Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P.

3.1/5 P'Relative_Deadline
Denotes a non-aliased component of the protected object P. This component is of type Ada.Real_Time.Time_Span and its value is the relative deadline of P. P'Relative_Deadline denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P.

4/5 The initial value of the attribute Priority is determined by the initial value of the priority of the protected object (see D.3), and can be changed by an assignment. The initial value of the attribute Relative_Deadline is determined by the initial value of the relative deadline of the protected object (see D.3), and can be changed by an assignment.

Dynamic Semantics

5/3 If the locking policy Ceiling_Locking (see D.3) is in effect, then the ceiling priority of a protected object *P* is set to the value of *P*'Priority at the end of each protected action of *P*.

6/3 If the locking policy Ceiling_Locking is in effect, then for a protected object *P* with either an Attach_Handler or Interrupt_Handler aspect specified for one of its procedures, a check is made that the value to be assigned to *P*'Priority is in the range System.Interrupt_Priority. If the check fails, Program_Error is raised.

Metrics

7/2 The implementation shall document the following metric:

8/2 • The difference in execution time of calls to the following procedures in protected object P:

9/2
```
protected P is
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority);
   procedure Set_Ceiling (Pr : System.Any_Priority);
end P;
```

```
protected body P is                                                    10/2
   procedure Do_Not_Set_Ceiling (Pr : System.Any_Priority) is
   begin
      null;
   end;
   procedure Set_Ceiling (Pr : System.Any_Priority) is
   begin
      P'Priority := Pr;
   end;
end P;
```

NOTE Since P'Priority is a normal variable, the value following an assignment to the attribute immediately reflects the new value even though its impact on the ceiling priority of P is postponed until completion of the protected action in which it is executed. 11/2

D.6 Preemptive Abort

This subclause specifies requirements on the immediacy with which an aborted construct is completed. 1/3

Dynamic Semantics

On a system with a single processor, an aborted construct is completed immediately at the first point that is outside the execution of an abort-deferred operation. 2

Documentation Requirements

On a multiprocessor, the implementation shall document any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. 3

Metrics

The implementation shall document the following metrics: 4

- The execution time, in processor clock cycles, that it takes for an abort_statement to cause the completion of the aborted task. This is measured in a situation where a task T2 preempts task T1 and aborts T1. T1 does not have any finalization code. T2 shall verify that T1 has terminated, by means of the Terminated attribute. 5
- On a multiprocessor, an upper bound in seconds, on the time that the completion of an aborted task can be delayed beyond the point that it is required for a single processor. 6
- An upper bound on the execution time of an asynchronous_select, in processor clock cycles. This is measured between a point immediately before a task T1 executes a protected operation Pr.Set that makes the condition of an entry_barrier Pr.Wait True, and the point where task T2 resumes execution immediately after an entry call to Pr.Wait in an asynchronous_select. T1 preempts T2 while T2 is executing the abortable part, and then blocks itself so that T2 can execute. The execution time of T1 is measured separately, and subtracted. 7/2
- An upper bound on the execution time of an asynchronous_select, in the case that no asynchronous transfer of control takes place. This is measured between a point immediately before a task executes the asynchronous_select with a nonnull abortable part, and the point where the task continues execution immediately after it. The execution time of the abortable part is subtracted. 8

Implementation Advice

Even though the abort_statement is included in the list of potentially blocking operations (see 9.5.1), it is recommended that this statement be implemented in a way that never requires the task executing the abort_statement to block. 9

10 On a multi-processor, the delay associated with aborting a task on another processor should be bounded; the implementation should use periodic polling, if necessary, to achieve this.

11 NOTE 1 Abortion does not change the active or base priority of the aborted task.

12 NOTE 2 Abortion cannot be more immediate than is allowed by the rules for deferral of abortion during finalization and in protected actions.

D.7 Tasking Restrictions

1/3 This subclause defines restrictions that can be used with a pragma Restrictions (see 13.12) to facilitate the construction of highly efficient tasking run-time systems.

Static Semantics

1.1/5 A scalar expression within a protected unit is said to be *pure-barrier-eligible* if it is one of the following:

1.2/5 - a static expression;

1.3/5 - a name that statically names (see 4.9) a scalar subcomponent of the immediately enclosing protected unit;

1.4/5 - a Count attribute_reference whose prefix statically denotes an entry declaration of the immediately enclosing unit;

1.5/5 - a call to a predefined relational operator or boolean logical operator (**and**, **or**, **xor**, **not**), where each operand is pure-barrier-eligible;

1.6/5 - a membership test whose *tested*_simple_expression is pure-barrier-eligible, and whose membership_choice_list meets the requirements for a static membership test (see 4.9);

1.7/5 - a short-circuit control form both of whose operands are pure-barrier-eligible;

1.8/5 - a conditional_expression all of whose conditions, *selecting*_expressions, and *dependent*_expressions are pure-barrier-eligible; or

1.9/5 - a pure-barrier-eligible expression enclosed in parentheses.

2 The following *restriction*_identifiers are language defined:

3/3 No_Task_Hierarchy
No task depends on a master other than the library-level master.

4/3 No_Nested_Finalization
Objects of a type that needs finalization (see 7.6) are declared only at library level. If an access type does not have library-level accessibility, then there are no allocators of the type where the type determined by the subtype_mark of the subtype_indication or qualified_expression needs finalization.

5/3 No_Abort_Statements
There are no abort_statements, and there is no use of a name denoting Task_Identification.Abort_Task.

6 No_Terminate_Alternatives
There are no selective_accepts with terminate_alternatives.

7 No_Task_Allocators
There are no allocators for task types or types containing task subcomponents.

7.1/3 In the case of an initialized allocator of an access type whose designated type is class-wide and limited, a check is made that the specific type of the allocated object has no task subcomponents. Program_Error is raised if this check fails.

No_Implicit_Heap_Allocations 8
There are no operations that implicitly require heap storage allocation to be performed by the implementation. The operations that implicitly require heap storage allocation are implementation defined.

No_Dynamic_Priorities 9/2
There are no semantic dependences on the package Dynamic_Priorities, and no occurrences of the attribute Priority.

No_Dynamic_Attachment 10/3
There is no use of a name denoting any of the operations defined in package Interrupts (Is_Reserved, Is_Attached, Current_Handler, Attach_Handler, Exchange_Handler, Detach_Handler, and Reference).

No_Dynamic_CPU_Assignment 10.1/4
No task has the CPU aspect specified to be a non-static expression. Each task (including the environment task) that has the CPU aspect specified as Not_A_Specific_CPU will be assigned to a particular implementation-defined CPU. The same is true for the environment task when the CPU aspect is not specified. Any other task without a CPU aspect will activate and execute on the same processor as its activating task.

No_Local_Protected_Objects 10.2/3
Protected objects are declared only at library level.

No_Local_Timing_Events 10.3/3
Timing_Events are declared only at library level.

No_Protected_Type_Allocators 10.4/2
There are no allocators for protected types or types containing protected type subcomponents.

In the case of an initialized allocator of an access type whose designated type is class-wide and limited, a check is made that the specific type of the allocated object has no protected subcomponents. Program_Error is raised if this check fails. 10.5/3

No_Relative_Delay 10.6/3
There are no delay_relative_statements, and there is no use of a name that denotes the Timing_Events.Set_Handler subprogram that has a Time_Span parameter.

No_Requeue_Statements 10.7/3
There are no requeue_statements.

No_Select_Statements 10.8/3
There are no select_statements.

No_Specific_Termination_Handlers 10.9/3
There is no use of a name denoting the Set_Specific_Handler and Specific_Handler subprograms in Task_Termination.

No_Tasks_Unassigned_To_CPU 10.10/4
The CPU aspect is specified for the environment task. No CPU aspect is specified to be statically equal to Not_A_Specific_CPU. If aspect CPU is specified (dynamically) to the value Not_A_Specific_CPU, then Program_Error is raised. If Set_CPU or Delay_Until_And_Set_CPU are called with the CPU parameter equal to Not_A_Specific_CPU, then Program_Error is raised.

Pure_Barriers 10.11/5
The Boolean expression in each protected entry barrier is pure-barrier-eligible.

10.12/5 Simple_Barriers
The Boolean expression in each entry barrier is either a static expression or a name that statically names (see 4.9) a subcomponent of the enclosing protected object.

11 The following *restriction_parameter*_identifiers are language defined:

12 Max_Select_Alternatives
Specifies the maximum number of alternatives in a selective_accept.

13 Max_Task_Entries
Specifies the maximum number of entries per task. The bounds of every entry family of a task unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static. A value of zero indicates that no rendezvous are possible.

14 Max_Protected_Entries
Specifies the maximum number of entries per protected type. The bounds of every entry family of a protected unit shall be static, or shall be defined by a discriminant of a subtype whose corresponding bound is static.

Dynamic Semantics

15/2 The following *restriction*_identifier is language defined:

15.1/2 No_Task_Termination
All tasks are nonterminating. It is implementation-defined what happens if a task attempts to terminate. If there is a fall-back handler (see C.7.3) set for the partition it should be called when the first task attempts to terminate.

16 The following *restriction_parameter*_identifiers are language defined:

17/1 Max_Storage_At_Blocking
Specifies the maximum portion (in storage elements) of a task's Storage_Size that can be retained by a blocked task. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; otherwise, the behavior is implementation defined.

18/1 Max_Asynchronous_Select_Nesting
Specifies the maximum dynamic nesting level of asynchronous_selects. A value of zero prevents the use of any asynchronous_select and, if a program contains an asynchronous_select, it is illegal. If an implementation chooses to detect a violation of this restriction for values other than zero, Storage_Error should be raised; otherwise, the behavior is implementation defined.

19/1 Max_Tasks Specifies the maximum number of task creations that may be executed over the lifetime of a partition, not counting the creation of the environment task. A value of zero prevents any task creation and, if a program contains a task creation, it is illegal. If an implementation chooses to detect a violation of this restriction, Storage_Error should be raised; otherwise, the behavior is implementation defined.

19.1/2 Max_Entry_Queue_Length
Max_Entry_Queue_Length defines the maximum number of calls that are queued on an entry. Violation of this restriction results in the raising of Program_Error at the point of the call or requeue.

19.2/3 No_Standard_Allocators_After_Elaboration
Specifies that an allocator using a standard storage pool (see 13.11) shall not occur within a parameterless library subprogram, nor within the handled_sequence_of_statements of a task body. For the purposes of this rule, an allocator of a type derived from a formal access type does not use a standard storage pool.

At run time, Storage_Error is raised if an allocator using a standard storage pool is evaluated after the elaboration of the library_items of the partition has completed. 19.3/3

It is implementation defined whether the use of pragma Restrictions results in a reduction in executable program size, storage requirements, or execution time. If possible, the implementation should provide quantitative descriptions of such effects for each restriction. 20

Implementation Advice

When feasible, the implementation should take advantage of the specified restrictions to produce a more efficient implementation. 21

NOTE The above Storage_Checks can be suppressed with pragma Suppress. 22

D.8 Monotonic Time

This subclause specifies a high-resolution, monotonic clock package. 1/3

Static Semantics

The following language-defined library package exists: 2

```
package Ada.Real_Time                                                          3/5
  with Nonblocking, Global => in out synchronized is

  type Time is private;                                                        4
  Time_First : constant Time;
  Time_Last : constant Time;
  Time_Unit : constant := implementation-defined-real-number;

  type Time_Span is private;                                                   5
  Time_Span_First : constant Time_Span;
  Time_Span_Last : constant Time_Span;
  Time_Span_Zero : constant Time_Span;
  Time_Span_Unit : constant Time_Span;

  Tick : constant Time_Span;                                                   6
  function Clock return Time;

  function "+" (Left : Time; Right : Time_Span) return Time;                   7
  function "+" (Left : Time_Span; Right : Time) return Time;
  function "-" (Left : Time; Right : Time_Span) return Time;
  function "-" (Left : Time; Right : Time) return Time_Span;

  function "<" (Left, Right : Time) return Boolean;                            8
  function "<="(Left, Right : Time) return Boolean;
  function ">" (Left, Right : Time) return Boolean;
  function ">="(Left, Right : Time) return Boolean;

  function "+" (Left, Right : Time_Span) return Time_Span;                     9
  function "-" (Left, Right : Time_Span) return Time_Span;
  function "-" (Right : Time_Span) return Time_Span;
  function "*" (Left : Time_Span; Right : Integer) return Time_Span;
  function "*" (Left : Integer; Right : Time_Span) return Time_Span;
  function "/" (Left, Right : Time_Span) return Integer;
  function "/" (Left : Time_Span; Right : Integer) return Time_Span;

  function "abs"(Right : Time_Span) return Time_Span;                          10

This paragraph was deleted.                                                    11/1

  function "<" (Left, Right : Time_Span) return Boolean;                       12
  function "<="(Left, Right : Time_Span) return Boolean;
  function ">" (Left, Right : Time_Span) return Boolean;
  function ">="(Left, Right : Time_Span) return Boolean;

  function To_Duration (TS : Time_Span) return Duration;                       13
  function To_Time_Span (D : Duration) return Time_Span;
```

14/2
```
      function Nanoseconds  (NS : Integer) return Time_Span;
      function Microseconds (US : Integer) return Time_Span;
      function Milliseconds (MS : Integer) return Time_Span;
      function Seconds      (S  : Integer) return Time_Span;
      function Minutes      (M  : Integer) return Time_Span;
```

15
```
      type Seconds_Count is range implementation-defined;
```

16
```
      procedure Split(T : in Time; SC : out Seconds_Count; TS : out Time_Span);
      function Time_Of(SC : Seconds_Count; TS : Time_Span) return Time;
```

17
```
   private
      ... -- not specified by the language
   end Ada.Real_Time;
```

18 In this Annex, *real time* is defined to be the physical time as observed in the external environment. The type Time is a *time type* as defined by 9.6; values of this type may be used in a delay_until_statement. Values of this type represent segments of an ideal time line. The set of values of the type Time corresponds one-to-one with an implementation-defined range of mathematical integers.

19 The Time value I represents the half-open real time interval that starts with E+I*Time_Unit and is limited by E+(I+1)*Time_Unit, where Time_Unit is an implementation-defined real number and E is an unspecified origin point, the *epoch*, that is the same for all values of the type Time. It is not specified by the language whether the time values are synchronized with any standard time reference. For example, E can correspond to the time of system initialization or it can correspond to the epoch of some time standard.

20 Values of the type Time_Span represent length of real time duration. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers. The Time_Span value corresponding to the integer I represents the real-time duration I*Time_Unit.

21 Time_First and Time_Last are the smallest and largest values of the Time type, respectively. Similarly, Time_Span_First and Time_Span_Last are the smallest and largest values of the Time_Span type, respectively.

22 A value of type Seconds_Count represents an elapsed time, measured in seconds, since the epoch.

Dynamic Semantics

23 Time_Unit is the smallest amount of real time representable by the Time type; it is expressed in seconds. Time_Span_Unit is the difference between two successive values of the Time type. It is also the smallest positive value of type Time_Span. Time_Unit and Time_Span_Unit represent the same real time duration. A *clock tick* is a real time interval during which the clock value (as observed by calling the Clock function) remains constant. Tick is the average length of such intervals.

24/2 The function To_Duration converts the value TS to a value of type Duration. Similarly, the function To_Time_Span converts the value D to a value of type Time_Span. For To_Duration, the result is rounded to the nearest value of type Duration (away from zero if exactly halfway between two values). If the result is outside the range of Duration, Constraint_Error is raised. For To_Time_Span, the value of D is first rounded to the nearest integral multiple of Time_Unit, away from zero if exactly halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the value is converted to the type Time_Span.

25 To_Duration(Time_Span_Zero) returns 0.0, and To_Time_Span(0.0) returns Time_Span_Zero.

26/2 The functions Nanoseconds, Microseconds, Milliseconds, Seconds, and Minutes convert the input parameter to a value of the type Time_Span. NS, US, MS, S, and M are interpreted as a number of nanoseconds, microseconds, milliseconds, seconds, and minutes respectively. The input parameter is first converted to seconds and rounded to the nearest integral multiple of Time_Unit, away from zero if exactly

halfway between two multiples. If the rounded value is outside the range of Time_Span, Constraint_Error is raised. Otherwise, the rounded value is converted to the type Time_Span.

The effects of the operators on Time and Time_Span are as for the operators defined for integer types. 27

The function Clock returns the amount of time since the epoch. 28

The effects of the Split and Time_Of operations are defined as follows, treating values of type Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split(T,SC,TS) is to set SC and TS to values such that T*Time_Unit = SC*1.0 + TS*Time_Unit, and 0.0 <= TS*Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the value T such that T*Time_Unit = SC*1.0 + TS*Time_Unit. 29

Implementation Requirements

The range of Time values shall be sufficient to uniquely represent the range of real times from program start-up to 50 years later. Tick shall be no greater than 1 millisecond. Time_Unit shall be less than or equal to 20 microseconds. 30

The value of Time_Span_First in seconds shall be no greater than –3600, and the value of Time_Span_Last in seconds shall be no less than 3600. 31/5

A *clock jump* is the difference between two successive distinct values of the clock (as observed by calling the Clock function). There shall be no backward clock jumps. 32

Documentation Requirements

The implementation shall document the values of Time_First, Time_Last, Time_Span_First, Time_Span_-Last, Time_Span_Unit, and Tick. 33

The implementation shall document the properties of the underlying time base used for the clock and for type Time, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used. 34

The implementation shall document whether or not there is any synchronization with external time references, and if such synchronization exists, the sources of synchronization information, the frequency of synchronization, and the synchronization method applied. 35

The implementation shall document any aspects of the external environment that can interfere with the clock behavior as defined in this subclause. 36/5

Metrics

For the purpose of the metrics defined in this subclause, real time is defined to be the International Atomic Time (TAI). 37/3

The implementation shall document the following metrics: 38

- An upper bound on the real-time duration of a clock tick. This is a value D such that if t1 and t2 are any real times such that t1 < t2 and $Clock_{t1} = Clock_{t2}$ then t2 – t1 <= D. 39
- An upper bound on the size of a clock jump. 40
- An upper bound on the *drift rate* of Clock with respect to real time. This is a real number D such that 41

  $E*(1–D) <= (Clock_{t+E} – Clock_t) <= E*(1+D)$
  provided that: $Clock_t + E*(1+D) <= Time\_Last$. 42

43 • where $Clock_t$ is the value of Clock at time t, and E is a real time duration not less than 24 hours. The value of E used for this metric shall be reported.

44 • An upper bound on the execution time of a call to the Clock function, in processor clock cycles.

45 • Upper bounds on the execution times of the operators of the types Time and Time_Span, in processor clock cycles.

Implementation Permissions

46/5 Implementations targeted to machines with word size smaller than 32 bits may omit support for the full range and granularity of the Time and Time_Span types.

Implementation Advice

47 When appropriate, implementations should provide configuration mechanisms to change the value of Tick.

48 It is recommended that Calendar.Clock and Real_Time.Clock be implemented as transformations of the same time base.

49 It is recommended that the "best" time base which exists in the underlying system be available to the application through Clock. "Best" may mean highest accuracy or largest range.

50/3 NOTE 1 The rules in this subclause do not imply that the implementation can protect the user from operator or installation errors that can result in the clock being set incorrectly.

51 NOTE 2 Time_Unit is the granularity of the Time type. In contrast, Tick represents the granularity of Real_Time.Clock. There is no requirement that these be the same.

D.9 Delay Accuracy

1/3 This subclause specifies performance requirements for the delay_statement. The rules apply both to delay_relative_statement and to delay_until_statement. Similarly, they apply equally to a simple delay_statement and to one which appears in a delay_alternative.

Dynamic Semantics

2 The effect of the delay_statement for Real_Time.Time is defined in terms of Real_Time.Clock:

3 • If C_1 is a value of Clock read before a task executes a delay_relative_statement with duration D, and C_2 is a value of Clock read after the task resumes execution following that delay_statement, then $C_2 - C_1 >= D$.

4 • If C is a value of Clock read after a task resumes execution following a delay_until_statement with Real_Time.Time value T, then C >= T.

5 A simple delay_statement with a negative or zero value for the expiration time does not cause the calling task to be blocked; it is nevertheless a potentially blocking operation (see 9.5.1).

6/3 When a delay_statement appears in a delay_alternative of a timed_entry_call the selection of the entry call is attempted, regardless of the specified expiration time. When a delay_statement appears in a select_alternative, and a call is queued on one of the open entries, the selection of that entry call proceeds, regardless of the value of the delay expression.

Documentation Requirements

7 The implementation shall document the minimum value of the delay expression of a delay_relative_statement that causes the task to actually be blocked.

The implementation shall document the minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked. 8

Metrics

The implementation shall document the following metrics: 9

- An upper bound on the execution time, in processor clock cycles, of a delay_relative_statement whose requested value of the delay expression is less than or equal to zero. 10
- An upper bound on the execution time, in processor clock cycles, of a delay_until_statement whose requested value of the delay expression is less than or equal to the value of Real_Time.Clock at the time of executing the statement. Similarly, for Calendar.Clock. 11
- An upper bound on the *lateness* of a delay_relative_statement, for a positive value of the delay expression, in a situation where the task has sufficient priority to preempt the processor as soon as it becomes ready, and can proceed without waiting for any other execution resources. The upper bound is expressed as a function of the value of the delay expression. The lateness is obtained by subtracting the value of the delay expression from the *actual duration*. The actual duration is measured from a point immediately before a task executes the delay_statement to a point immediately after the task resumes execution following this statement. 12/5
- An upper bound on the lateness of a delay_until_statement, in a situation where the value of the requested expiration time is after the time the task begins executing the statement, the task has sufficient priority to preempt the processor as soon as it becomes ready, and it can proceed without waiting for any other execution resources. The upper bound is expressed as a function of the difference between the requested expiration time and the clock value at the time the statement begins execution. The lateness of a delay_until_statement is obtained by subtracting the requested expiration time from the real time that the task resumes execution following this statement. 13/5

D.10 Synchronous Task Control

This subclause describes a language-defined private semaphore (suspension object), which can be used for *two-stage suspend* operations and as a simple building block for implementing higher-level queues. 1/3

Static Semantics

The following language-defined package exists: 2

```
package Ada.Synchronous_Task_Control                                    3/5
  with Preelaborate, Nonblocking, Global => in out synchronized is

  type Suspension_Object is limited private;                            4/5
  procedure Set_True(S : in out Suspension_Object);
  procedure Set_False(S : in out Suspension_Object);
  function Current_State(S : Suspension_Object) return Boolean;
  procedure Suspend_Until_True(S : in out Suspension_Object)
     with Nonblocking => False;
private
     ... -- not specified by the language
end Ada.Synchronous_Task_Control;
```

The type Suspension_Object is a by-reference type. 5

The following language-defined package exists: 5.1/3

5.2/5
```
with Ada.Real_Time;
package Ada.Synchronous_Task_Control.EDF
   with Nonblocking, Global => in out synchronized is
   procedure Suspend_Until_True_And_Set_Deadline
      (S  : in out Suspension_Object;
       TS : in     Ada.Real_Time.Time_Span)
      with Nonblocking => False;
end Ada.Synchronous_Task_Control.EDF;
```

Dynamic Semantics

6/2 An object of the type Suspension_Object has two visible states: True and False. Upon initialization, its value is set to False.

7/2 The operations Set_True and Set_False are atomic with respect to each other and with respect to Suspend_Until_True; they set the state to True and False respectively.

8 Current_State returns the current state of the object.

9/2 The procedure Suspend_Until_True blocks the calling task until the state of the object S is True; at that point the task becomes ready and the state of the object becomes False.

10/5 Program_Error is raised upon calling Suspend_Until_True if another task is already waiting on that suspension object.

10.1/5 The procedure Suspend_Until_True_And_Set_Deadline blocks the calling task until the state of the object S is True; at that point the task becomes ready with a deadline of Ada.Real_Time.Clock + TS, and the state of the object becomes False. Program_Error is raised upon calling Suspend_Until_True_And_Set_Deadline if another task is already waiting on that suspension object.

Bounded (Run-Time) Errors

10.2/5 It is a bounded error for two or more tasks to call Suspend_Until_True on the same Suspension_Object concurrently. For each task, Program_Error can be raised, the task can proceed without suspending, or the task can suspend, potentially indefinitely. The state of the suspension object can end up either True or False.

Implementation Requirements

11 The implementation is required to allow the calling of Set_False and Set_True during any protected action, even one that has its ceiling priority in the Interrupt_Priority range.

12/3 NOTE More complex schemes, such as setting the deadline relative to when Set_True is called, can be programmed using a protected object.

D.10.1 Synchronous Barriers

1/3 This subclause introduces a language-defined package to synchronously release a group of tasks after the number of blocked tasks reaches a specified count value.

Static Semantics

2/3 The following language-defined library package exists:

3/5
```
package Ada.Synchronous_Barriers
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/3
```
   subtype Barrier_Limit is Positive range 1 .. implementation-defined;
```

5/3
```
   type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is limited
private;
```

```
   procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
                              Notified    :    out Boolean)
      with Nonblocking => False;
private
   -- not specified by the language
end Ada.Synchronous_Barriers;
```

Type Synchronous_Barrier needs finalization (see 7.6).

Dynamic Semantics

Each call to Wait_For_Release blocks the calling task until the number of blocked tasks associated with the Synchronous_Barrier object is equal to Release_Threshold, at which time all blocked tasks are released. Notified is set to True for one of the released tasks, and set to False for all other released tasks.

The mechanism for determining which task sets Notified to True is implementation defined.

Once all tasks have been released, a Synchronous_Barrier object may be reused to block another Release_Threshold number of tasks.

As the first step of the finalization of a Synchronous_Barrier, each blocked task is unblocked and Program_Error is raised at the place of the call to Wait_For_Release.

It is implementation defined whether an abnormal task which is waiting on a Synchronous_Barrier object is aborted immediately or aborted when the tasks waiting on the object are released.

This paragraph was deleted.

Bounded (Run-Time) Errors

It is a bounded error to call Wait_For_Release on a Synchronous_Barrier object after that object is finalized. If the error is detected, Program_Error is raised. Otherwise, the call proceeds normally, which may leave a task blocked forever.

D.11 Asynchronous Task Control

This subclause introduces a language-defined package to do asynchronous suspend/resume on tasks. It uses a conceptual *held priority* value to represent the task's *held* state.

Static Semantics

The following language-defined library package exists:

```
with Ada.Task_Identification;
package Ada.Asynchronous_Task_Control
  with Preelaborate, Nonblocking, Global => in out synchronized is
  procedure Hold(T : in Ada.Task_Identification.Task_Id);
  procedure Continue(T : in Ada.Task_Identification.Task_Id);
  function Is_Held(T : Ada.Task_Identification.Task_Id)
   return Boolean;
end Ada.Asynchronous_Task_Control;
```

Dynamic Semantics

After the Hold operation has been applied to a task, the task becomes *held*. For each processor there is a conceptual *idle task*, which is always ready. The base priority of the idle task is below System.Any_-Priority'First. The *held priority* is a constant of the type Integer whose value is below the base priority of the idle task.

For any priority below System.Any_Priority'First, the task dispatching policy is FIFO_Within_Priorities.

5/2 The Hold operation sets the state of T to held. For a held task, the active priority is reevaluated as if the base priority of the task were the held priority.

6/2 The Continue operation resets the state of T to not-held; its active priority is then reevaluated as determined by the task dispatching policy associated with its base priority.

7 The Is_Held function returns True if and only if T is in the held state.

8 As part of these operations, a check is made that the task identified by T is not terminated. Tasking_Error is raised if the check fails. Program_Error is raised if the value of T is Null_Task_Id.

Erroneous Execution

9 If any operation in this package is called with a parameter T that specifies a task object that no longer exists, the execution of the program is erroneous.

Implementation Permissions

10/5 An implementation may omit support for Asynchronous_Task_Control if it is infeasible to support it in the target environment.

11 NOTE 1 It is a consequence of the priority rules that held tasks cannot be dispatched on any processor in a partition (unless they are inheriting priorities) since their priorities are defined to be below the priority of any idle task.

12 NOTE 2 The effect of calling Get_Priority and Set_Priority on a Held task is the same as on any other task.

13 NOTE 3 Calling Hold on a held task or Continue on a non-held task has no effect.

14 NOTE 4 The rules affecting queuing are derived from the above rules, in addition to the normal priority rules:

15 • When a held task is on the ready queue, its priority is so low as to never reach the top of the queue as long as there are other tasks on that queue.

16 • If a task is executing in a protected action, inside a rendezvous, or is inheriting priorities from other sources (e.g. when activated), it continues to execute until it is no longer executing the corresponding construct.

17 • If a task becomes held while waiting (as a caller) for a rendezvous to complete, the active priority of the accepting task is not affected.

18/1 • If a task becomes held while waiting in a selective_accept, and an entry call is issued to one of the open entries, the corresponding accept_alternative executes. When the rendezvous completes, the active priority of the accepting task is lowered to the held priority (unless it is still inheriting from other sources), and the task does not execute until another Continue.

19 • The same holds if the held task is the only task on a protected entry queue whose barrier becomes open. The corresponding entry body executes.

D.12 Other Optimizations and Determinism Rules

1/3 This subclause describes various requirements for improving the response and determinism in a real-time system.

Implementation Requirements

2 If the implementation blocks interrupts (see C.3) not as a result of direct user action (e.g. an execution of a protected action) there shall be an upper bound on the duration of this blocking.

3 The implementation shall recognize entry-less protected types. The overhead of acquiring the execution resource of an object of such a type (see 9.5.1) shall be minimized. In particular, there should not be any overhead due to evaluating entry_barrier conditions.

Unchecked_Deallocation shall be supported for terminated tasks that are designated by access types, and shall have the effect of releasing all the storage associated with the task. This includes any run-time system or heap storage that has been implicitly allocated for the task by the implementation. 4

Documentation Requirements

The implementation shall document the upper bound on the duration of interrupt blocking caused by the implementation. If this is different for different interrupts or interrupt priority levels, it should be documented for each case. 5

Metrics

The implementation shall document the following metric: 6

- The overhead associated with obtaining a mutual-exclusive access to an entry-less protected object. This shall be measured in the following way: 7

  For a protected object of the form: 8

```
protected Lock is
   procedure Set;
   function Read return Boolean;
private
   Flag : Boolean := False;
end Lock;
```

```
protected body Lock is
   procedure Set is
   begin
      Flag := True;
   end Set;
   function Read return Boolean
   begin
      return Flag;
   end Read;
end Lock;
```

  The execution time, in processor clock cycles, of a call to Set. This shall be measured between the point just before issuing the call, and the point just after the call completes. The function Read shall be called later to verify that Set was indeed called (and not optimized away). The calling task shall have sufficiently high priority as to not be preempted during the measurement period. The protected object shall have sufficiently high ceiling priority to allow the task to call Set. 11

  For a multiprocessor, if supported, the metric shall be reported for the case where no contention (on the execution resource) exists from tasks executing on other processors. 12

D.13 The Ravenscar and Jorvik Profiles

This subclause defines the Ravenscar and Jorvik profiles. 1/5

Paragraphs 2 and 3 were moved to 13.12, "Pragma Restrictions and Pragma Profile".

Legality Rules

The *profile*_identifier Ravenscar and *profile*_identifier Jorvik are usage profiles (see 13.12). For usage profiles Ravenscar and Jorvik, there shall be no *profile*_pragma_argument_associations. 4/3

Static Semantics

The usage profile Ravenscar is equivalent to the following set of pragmas: 5/3

6/4
```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
               No_Abort_Statements,
               No_Dynamic_Attachment,
               No_Dynamic_CPU_Assignment,
               No_Dynamic_Priorities,
               No_Implicit_Heap_Allocations,
               No_Local_Protected_Objects,
               No_Local_Timing_Events,
               No_Protected_Type_Allocators,
               No_Relative_Delay,
               No_Requeue_Statements,
               No_Select_Statements,
               No_Specific_Termination_Handlers,
               No_Task_Allocators,
               No_Task_Hierarchy,
               No_Task_Termination,
               Simple_Barriers,
               Max_Entry_Queue_Length => 1,
               Max_Protected_Entries => 1,
               Max_Task_Entries => 0,
               No_Dependence => Ada.Asynchronous_Task_Control,
               No_Dependence => Ada.Calendar,
               No_Dependence => Ada.Execution_Time.Group_Budgets,
               No_Dependence => Ada.Execution_Time.Timers,
               No_Dependence => Ada.Synchronous_Barriers,
               No_Dependence => Ada.Task_Attributes,
               No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

6.1/5 The usage profile Jorvik is equivalent to the following set of pragmas:

6.2/5
```
pragma Task_Dispatching_Policy (FIFO_Within_Priorities);
pragma Locking_Policy (Ceiling_Locking);
pragma Detect_Blocking;
pragma Restrictions (
               No_Abort_Statements,
               No_Dynamic_Attachment,
               No_Dynamic_CPU_Assignment,
               No_Dynamic_Priorities,
               No_Local_Protected_Objects,
               No_Local_Timing_Events,
               No_Protected_Type_Allocators,
               No_Requeue_Statements,
               No_Select_Statements,
               No_Specific_Termination_Handlers,
               No_Task_Allocators,
               No_Task_Hierarchy,
               No_Task_Termination,
               Pure_Barriers,
               Max_Task_Entries => 0,
               No_Dependence => Ada.Asynchronous_Task_Control,
               No_Dependence => Ada.Execution_Time.Group_Budgets,
               No_Dependence => Ada.Execution_Time.Timers,
               No_Dependence => Ada.Task_Attributes,
               No_Dependence => System.Multiprocessors.Dispatching_Domains);
```

Paragraph 7 and 8 were deleted.

Implementation Advice

9/5 On a multiprocessor system, an implementation should support a fully partitioned approach if one of these profiles is specified. Each processor should have separate and disjoint ready queues.

NOTE 1 For the Ravenscar profile, the effect of the restriction Max_Entry_Queue_Length => 1 applies only to protected entry queues due to the accompanying restriction Max_Task_Entries => 0. The restriction Max_Entry_Queue_Length is not applied by the Jorvik profile. 10/5

NOTE 2 When the Ravenscar or Jorvik profile is in effect (via the effect of the No_Dynamic_CPU_Assignment restriction), all of the tasks in the partition will execute on a single CPU unless the programmer explicitly uses aspect CPU to specify the CPU assignments for tasks. The use of multiple CPUs requires care, as many guarantees of single CPU scheduling no longer apply. 11/5

NOTE 3 12/5

NOTE 4 Any unit that meets the requirements of the Ravenscar profile also meets the requirements of the Jorvik profile. 13/5

D.14 Execution Time

This subclause describes a language-defined package to measure execution time. 1/3

Static Semantics

The following language-defined library package exists: 2/2

```
with Ada.Task_Identification;                                                    3/5
with Ada.Real_Time; use Ada.Real_Time;
package Ada.Execution_Time
   with Nonblocking, Global => in out synchronized is

   type CPU_Time is private;                                                     4/2
   CPU_Time_First : constant CPU_Time;
   CPU_Time_Last  : constant CPU_Time;
   CPU_Time_Unit  : constant := implementation-defined-real-number;
   CPU_Tick : constant Time_Span;

   function Clock                                                               5/2
     (T : Ada.Task_Identification.Task_Id
          := Ada.Task_Identification.Current_Task)
     return CPU_Time;

   function "+"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;          6/2
   function "+"  (Left : Time_Span; Right : CPU_Time) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : Time_Span) return CPU_Time;
   function "-"  (Left : CPU_Time; Right : CPU_Time)  return Time_Span;

   function "<"  (Left, Right : CPU_Time) return Boolean;                       7/2
   function "<=" (Left, Right : CPU_Time) return Boolean;
   function ">"  (Left, Right : CPU_Time) return Boolean;
   function ">=" (Left, Right : CPU_Time) return Boolean;

   procedure Split                                                              8/2
     (T : in CPU_Time; SC : out Seconds_Count; TS : out Time_Span);

   function Time_Of (SC : Seconds_Count;                                        9/2
                     TS : Time_Span := Time_Span_Zero) return CPU_Time;

   Interrupt_Clocks_Supported : constant Boolean := implementation-defined;     9.1/3

   Separate_Interrupt_Clocks_Supported : constant Boolean :=                    9.2/3
     implementation-defined;

   function Clock_For_Interrupts return CPU_Time;                               9.3/3

private                                                                         10/2
   ... -- not specified by the language
end Ada.Execution_Time;
```

The *execution time* or CPU time of a given task is defined as the time spent by the system executing that task, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. The Boolean constant Interrupt_Clocks_Supported is set to True if the implementation separately accounts for the execution time of interrupt handlers. If it is set to False it is implementation defined which task, if any, is charged the execution time that is consumed 11/3

by interrupt handlers. The Boolean constant Separate_Interrupt_Clocks_Supported is set to True if the implementation separately accounts for the execution time of individual interrupt handlers (see D.14.3).

12/2 The type CPU_Time represents the execution time of a task. The set of values of this type corresponds one-to-one with an implementation-defined range of mathematical integers.

13/2 The CPU_Time value I represents the half-open execution-time interval that starts with I*CPU_Time_Unit and is limited by (I+1)*CPU_Time_Unit, where CPU_Time_Unit is an implementation-defined real number. For each task, the execution time value is set to zero at the creation of the task.

14/2 CPU_Time_First and CPU_Time_Last are the smallest and largest values of the CPU_Time type, respectively.

14.1/3 The execution time value for the function Clock_For_Interrupts is initialized to zero.

Dynamic Semantics

15/2 CPU_Time_Unit is the smallest amount of execution time representable by the CPU_Time type; it is expressed in seconds. A *CPU clock tick* is an execution time interval during which the clock value (as observed by calling the Clock function) remains constant. CPU_Tick is the average length of such intervals.

16/2 The effects of the operators on CPU_Time and Time_Span are as for the operators defined for integer types.

17/2 The function Clock returns the current execution time of the task identified by T; Tasking_Error is raised if that task has terminated; Program_Error is raised if the value of T is Task_Identification.Null_Task_Id.

18/2 The effects of the Split and Time_Of operations are defined as follows, treating values of type CPU_Time, Time_Span, and Seconds_Count as mathematical integers. The effect of Split (T, SC, TS) is to set SC and TS to values such that T*CPU_Time_Unit = SC*1.0 + TS*CPU_Time_Unit, and 0.0 <= TS*CPU_Time_Unit < 1.0. The value returned by Time_Of(SC,TS) is the execution-time value T such that T*CPU_Time_Unit=SC*1.0 + TS*CPU_Time_Unit.

18.1/3 The function Clock_For_Interrupts returns the total cumulative time spent executing within all interrupt handlers. This time is not allocated to any task execution time clock. If Interrupt_Clocks_Supported is set to False the function raises Program_Error.

Erroneous Execution

19/2 For a call of Clock, if the task identified by T no longer exists, the execution of the program is erroneous.

Implementation Requirements

20/2 The range of CPU_Time values shall be sufficient to uniquely represent the range of execution times from the task start-up to 50 years of execution time later. CPU_Tick shall be no greater than 1 millisecond.

Documentation Requirements

21/2 The implementation shall document the values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick.

22/2 The implementation shall document the properties of the underlying mechanism used to measure execution times, such as the range of values supported and any relevant aspects of the underlying hardware or operating system facilities used.

Metrics

The implementation shall document the following metrics: 23/2

- An upper bound on the execution-time duration of a clock tick. This is a value D such that if t1 and t2 are any execution times of a given task such that t1 < t2 and $Clock_{t1} = Clock_{t2}$ then t2 – t1 <= D. 24/2
- An upper bound on the size of a clock jump. A clock jump is the difference between two successive distinct values of an execution-time clock (as observed by calling the Clock function with the same Task_Id). 25/2
- An upper bound on the execution time of a call to the Clock function, in processor clock cycles. 26/2
- Upper bounds on the execution times of the operators of the type CPU_Time, in processor clock cycles. 27/2

Implementation Permissions

Implementations targeted to machines with word size smaller than 32 bits may omit support for the full range and granularity of the CPU_Time type. 28/5

Implementation Advice

When appropriate, implementations should provide configuration mechanisms to change the value of CPU_Tick. 29/2

D.14.1 Execution Time Timers

This subclause describes a language-defined package that provides a facility for calling a handler when a task has used a defined amount of CPU time. 1/3

Static Semantics

The following language-defined library package exists: 2/2

```
with System;                                                                    3/5
package Ada.Execution_Time.Timers
   with Nonblocking, Global => in out synchronized is

   type Timer (T : not null access constant                                     4/2
                      Ada.Task_Identification.Task_Id) is
      tagged limited private;

   type Timer_Handler is                                                        5/5
      access protected procedure (TM : in out Timer)
      with Nonblocking => False;

   Min_Handler_Ceiling : constant System.Any_Priority :=                        6/2
   implementation-defined;

   procedure Set_Handler (TM      : in out Timer;                               7/2
                          In_Time : in Time_Span;
                          Handler : in Timer_Handler);
   procedure Set_Handler (TM      : in out Timer;
                          At_Time : in CPU_Time;
                          Handler : in Timer_Handler);
   function Current_Handler (TM : Timer) return Timer_Handler;
   procedure Cancel_Handler (TM        : in out Timer;
                             Cancelled :    out Boolean);

   function Time_Remaining (TM : Timer) return Time_Span;                       8/2

   Timer_Resource_Error : exception;                                            9/2
```

10/2
```
   private
      ... -- not specified by the language
   end Ada.Execution_Time.Timers;
```

11/2 The type Timer represents an execution-time event for a single task and is capable of detecting execution-time overruns. The access discriminant T identifies the task concerned. The type Timer needs finalization (see 7.6).

12/2 An object of type Timer is said to be *set* if it is associated with a nonnull value of type Timer_Handler and *cleared* otherwise. All Timer objects are initially cleared.

13/2 The type Timer_Handler identifies a protected procedure to be executed by the implementation when the timer expires. Such a protected procedure is called a *handler*.

Dynamic Semantics

14/2 When a Timer object is created, or upon the first call of a Set_Handler procedure with the timer as parameter, the resources required to operate an execution-time timer based on the associated execution-time clock are allocated and initialized. If this operation would exceed the available resources, Timer_Resource_Error is raised.

15/3 The procedures Set_Handler associate the handler Handler with the timer TM: if Handler is **null**, the timer is cleared; otherwise, it is set. The first procedure Set_Handler loads the timer TM with an interval specified by the Time_Span parameter. In this mode, the timer TM *expires* when the execution time of the task identified by TM.T.**all** has increased by In_Time; if In_Time is less than or equal to zero, the timer expires immediately. The second procedure Set_Handler loads the timer TM with the absolute value specified by At_Time. In this mode, the timer TM expires when the execution time of the task identified by TM.T.**all** reaches At_Time; if the value of At_Time has already been reached when Set_Handler is called, the timer expires immediately.

16/2 A call of a procedure Set_Handler for a timer that is already set replaces the handler and the (absolute or relative) execution time; if Handler is not **null**, the timer remains set.

17/2 When a timer expires, the associated handler is executed, passing the timer as parameter. The initial action of the execution of the handler is to clear the event.

18/3 The function Current_Handler returns the handler associated with the timer TM if that timer is set; otherwise, it returns **null**.

19/3 The procedure Cancel_Handler clears the timer if it is set. Cancelled is assigned True if the timer was set prior to it being cleared; otherwise, it is assigned False.

20/3 The function Time_Remaining returns the execution time interval that remains until the timer TM would expire, if that timer is set; otherwise, it returns Time_Span_Zero.

21/2 The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked.

22/2 As part of the finalization of an object of type Timer, the timer is cleared.

23/2 For all the subprograms defined in this package, Tasking_Error is raised if the task identified by TM.T.**all** has terminated, and Program_Error is raised if the value of TM.T.**all** is Task_Identification.Null_Task_Id.

24/2 An exception propagated from a handler invoked as part of the expiration of a timer has no effect.

Erroneous Execution

For a call of any of the subprograms defined in this package, if the task identified by TM.T.**all** no longer exists, the execution of the program is erroneous. 25/2

Implementation Requirements

For a given Timer object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timer object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler. 26/2

When an object of type Timer is finalized, the system resources used by the timer shall be deallocated. 27/2

Implementation Permissions

Implementations may limit the number of timers that can be defined for each task. If this limit is exceeded, then Timer_Resource_Error is raised. 28/3

NOTE A Timer_Handler can be associated with several Timer objects. 29/2

D.14.2 Group Execution Time Budgets

This subclause describes a language-defined package to assign execution time budgets to groups of tasks. 1/3

Static Semantics

The following language-defined library package exists: 2/2

```
with System;                                                                    3/5
with System.Multiprocessors;
package Ada.Execution_Time.Group_Budgets
  with Nonblocking, Global => in out synchronized is

  type Group_Budget(CPU : System.Multiprocessors.CPU :=                         4/3
                          System.Multiprocessors.CPU'First)
    is tagged limited private;

  type Group_Budget_Handler is access                                           5/5
       protected procedure (GB : in out Group_Budget)
       with Nonblocking => False;

  type Task_Array is array (Positive range <>) of                               6/2
                                 Ada.Task_Identification.Task_Id;

  Min_Handler_Ceiling : constant System.Any_Priority :=                         7/2
    implementation-defined;

  procedure Add_Task (GB : in out Group_Budget;                                 8/2
                      T  : in Ada.Task_Identification.Task_Id);
  procedure Remove_Task (GB: in out Group_Budget;
                         T  : in Ada.Task_Identification.Task_Id);
  function Is_Member (GB : Group_Budget;
                      T : Ada.Task_Identification.Task_Id) return Boolean;
  function Is_A_Group_Member
     (T : Ada.Task_Identification.Task_Id) return Boolean;
  function Members (GB : Group_Budget) return Task_Array;

  procedure Replenish (GB : in out Group_Budget; To : in Time_Span);            9/2
  procedure Add (GB : in out Group_Budget; Interval : in Time_Span);
  function Budget_Has_Expired (GB : Group_Budget) return Boolean;
  function Budget_Remaining (GB : Group_Budget) return Time_Span;

  procedure Set_Handler (GB      : in out Group_Budget;                         10/2
                         Handler : in Group_Budget_Handler);
  function Current_Handler (GB : Group_Budget)
     return Group_Budget_Handler;
  procedure Cancel_Handler (GB        : in out Group_Budget;
                            Cancelled : out Boolean);
```

11/2
```
   Group_Budget_Error : exception;
```

12/2
```
private
   -- not specified by the language
end Ada.Execution_Time.Group_Budgets;
```

13/2 The type Group_Budget represents an execution time budget to be used by a group of tasks. The type Group_Budget needs finalization (see 7.6). A task can belong to at most one group. Tasks of any priority can be added to a group.

14/2 An object of type Group_Budget has an associated nonnegative value of type Time_Span known as its *budget*, which is initially Time_Span_Zero. The type Group_Budget_Handler identifies a protected procedure to be executed by the implementation when the budget is *exhausted*, that is, reaches zero. Such a protected procedure is called a *handler*.

15/2 An object of type Group_Budget also includes a handler, which is a value of type Group_Budget_Handler. The handler of the object is said to be *set* if it is not null and *cleared* otherwise. The handler of all Group_Budget objects is initially cleared.

Dynamic Semantics

16/2 The procedure Add_Task adds the task identified by T to the group GB; if that task is already a member of some other group, Group_Budget_Error is raised.

17/2 The procedure Remove_Task removes the task identified by T from the group GB; if that task is not a member of the group GB, Group_Budget_Error is raised. After successful execution of this procedure, the task is no longer a member of any group.

18/3 The function Is_Member returns True if the task identified by T is a member of the group GB; otherwise, it returns False.

19/3 The function Is_A_Group_Member returns True if the task identified by T is a member of some group; otherwise, it returns False.

20/2 The function Members returns an array of values of type Task_Identification.Task_Id identifying the members of the group GB. The order of the components of the array is unspecified.

21/3 The procedure Replenish loads the group budget GB with To as the Time_Span value. The exception Group_Budget_Error is raised if the Time_Span value To is nonpositive. Any execution on CPU of any member of the group of tasks results in the budget counting down, unless exhausted. When the budget becomes exhausted (reaches Time_Span_Zero), the associated handler is executed if the handler of group budget GB is set. Nevertheless, the tasks continue to execute.

22/2 The procedure Add modifies the budget of the group GB. A positive value for Interval increases the budget. A negative value for Interval reduces the budget, but never below Time_Span_Zero. A zero value for Interval has no effect. A call of procedure Add that results in the value of the budget going to Time_Span_Zero causes the associated handler to be executed if the handler of the group budget GB is set.

23/3 The function Budget_Has_Expired returns True if the budget of group GB is exhausted (equal to Time_Span_Zero); otherwise, it returns False.

24/2 The function Budget_Remaining returns the remaining budget for the group GB. If the budget is exhausted it returns Time_Span_Zero. This is the minimum value for a budget.

25/3 The procedure Set_Handler associates the handler Handler with the Group_Budget GB: if Handler is **null**, the handler of Group_Budget is cleared; otherwise, it is set.

A call of Set_Handler for a Group_Budget that already has a handler set replaces the handler; if Handler is not **null**, the handler for Group_Budget remains set. 26/2

The function Current_Handler returns the handler associated with the group budget GB if the handler for that group budget is set; otherwise, it returns **null**. 27/3

The procedure Cancel_Handler clears the handler for the group budget if it is set. Cancelled is assigned True if the handler for the group budget was set prior to it being cleared; otherwise, it is assigned False. 28/3

The constant Min_Handler_Ceiling is the minimum ceiling priority required for a protected object with a handler to ensure that no ceiling violation will occur when that handler is invoked. 29/2

The precision of the accounting of task execution time to a Group_Budget is the same as that defined for execution-time clocks from the parent package. 30/2

As part of the finalization of an object of type Group_Budget all member tasks are removed from the group identified by that object. 31/2

If a task is a member of a Group_Budget when it terminates, then as part of the finalization of the task it is removed from the group. 32/3

For all the operations defined in this package, Tasking_Error is raised if the task identified by T has terminated, and Program_Error is raised if the value of T is Task_Identification.Null_Task_Id. 33/2

An exception propagated from a handler invoked when the budget of a group of tasks becomes exhausted has no effect. 34/2

Erroneous Execution

For a call of any of the subprograms defined in this package, if the task identified by T no longer exists, the execution of the program is erroneous. 35/2

Implementation Requirements

For a given Group_Budget object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Group_Budget object. The replacement of a handler, by a call of Set_Handler, shall be performed atomically with respect to the execution of the handler. 36/2

NOTE 1 Clearing or setting of the handler of a group budget does not change the current value of the budget. Exhaustion or loading of a budget does not change whether the handler of the group budget is set or cleared. 37/2

NOTE 2 A Group_Budget_Handler can be associated with several Group_Budget objects. 38/2

D.14.3 Execution Time of Interrupt Handlers

This subclause describes a language-defined package to measure the execution time of interrupt handlers. 1/3

Static Semantics

The following language-defined library package exists: 2/3

```
with Ada.Interrupts;                                                    3/5
package Ada.Execution_Time.Interrupts
   with Nonblocking, Global => in out synchronized is
   function Clock (Interrupt : Ada.Interrupts.Interrupt_Id)
        return CPU_Time;
   function Supported (Interrupt : Ada.Interrupts.Interrupt_Id)
        return Boolean;
end Ada.Execution_Time.Interrupts;
```

4/3 The execution time or CPU time of a given interrupt Interrupt is defined as the time spent by the system executing interrupt handlers identified by Interrupt, including the time spent executing run-time or system services on its behalf. The mechanism used to measure execution time is implementation defined. Time spent executing interrupt handlers is distinct from time spent executing any task.

5/3 For each interrupt, the execution time value is initially set to zero.

Dynamic Semantics

6/3 The function Clock returns the current cumulative execution time of the interrupt identified by Interrupt. If Separate_Interrupt_Clocks_Supported is set to False the function raises Program_Error.

7/3 The function Supported returns True if the implementation is monitoring the execution time of the interrupt identified by Interrupt; otherwise, it returns False. For any Interrupt_Id Interrupt for which Supported(Interrupt) returns False, the function Clock(Interrupt) will return a value equal to Ada.Execution_Time.Time_Of(0).

D.15 Timing Events

1/5 This subclause describes a language-defined package to allow user-defined protected procedures to be executed at a specified time without the use of a task or a delay statement.

Static Semantics

2/2 The following language-defined library package exists:

3/5
```
package Ada.Real_Time.Timing_Events
  with Nonblocking, Global => in out synchronized is
```

4/5
```
  type Timing_Event is tagged limited private;
  type Timing_Event_Handler
       is access protected procedure (Event : in out Timing_Event)
       with Nonblocking => False;
```

5/2
```
  procedure Set_Handler (Event   : in out Timing_Event;
                         At_Time : in Time;
                         Handler : in Timing_Event_Handler);
  procedure Set_Handler (Event   : in out Timing_Event;
                         In_Time : in Time_Span;
                         Handler : in Timing_Event_Handler);
  function Current_Handler (Event : Timing_Event)
       return Timing_Event_Handler;
  procedure Cancel_Handler (Event     : in out Timing_Event;
                            Cancelled : out Boolean);
```

6/2
```
  function Time_Of_Event (Event : Timing_Event) return Time;
```

7/2
```
private
  ... -- not specified by the language
end Ada.Real_Time.Timing_Events;
```

8/2 The type Timing_Event represents a time in the future when an event is to occur. The type Timing_Event needs finalization (see 7.6).

9/2 An object of type Timing_Event is said to be *set* if it is associated with a nonnull value of type Timing_Event_Handler and *cleared* otherwise. All Timing_Event objects are initially cleared.

10/2 The type Timing_Event_Handler identifies a protected procedure to be executed by the implementation when the timing event occurs. Such a protected procedure is called a *handler*.

Dynamic Semantics

The procedures Set_Handler associate the handler Handler with the event Event: if Handler is **null**, the event is cleared; otherwise, it is set. The first procedure Set_Handler sets the execution time for the event to be At_Time. The second procedure Set_Handler sets the execution time for the event to be Real_Time.Clock + In_Time. 11/3

A call of a procedure Set_Handler for an event that is already set replaces the handler and the time of execution; if Handler is not **null**, the event remains set. 12/2

As soon as possible after the time set for the event, the handler is executed, passing the event as parameter. The handler is only executed if the timing event is in the set state at the time of execution. The initial action of the execution of the handler is to clear the event. 13/2

If the Ceiling_Locking policy (see D.3) is in effect when a procedure Set_Handler is called, a check is made that the ceiling priority of Handler.**all** is Interrupt_Priority'Last. If the check fails, Program_Error is raised. 14/2

If a procedure Set_Handler is called with zero or negative In_Time or with At_Time indicating a time in the past, then the handler is executed as soon as possible after the completion of the call of Set_Handler. 15/3

The function Current_Handler returns the handler associated with the event Event if that event is set; otherwise, it returns **null**. 16/3

The procedure Cancel_Handler clears the event if it is set. Cancelled is assigned True if the event was set prior to it being cleared; otherwise, it is assigned False. 17/3

The function Time_Of_Event returns the time of the event if the event is set; otherwise, it returns Real_Time.Time_First. 18/3

As part of the finalization of an object of type Timing_Event, the Timing_Event is cleared. 19/2

If several timing events are set for the same time, they are executed in FIFO order of being set. 20/2

An exception propagated from a handler invoked by a timing event has no effect. 21/2

Implementation Requirements

For a given Timing_Event object, the implementation shall perform the operations declared in this package atomically with respect to any of these operations on the same Timing_Event object. The replacement of a handler by a call of Set_Handler shall be performed atomically with respect to the execution of the handler. 22/2

Metrics

The implementation shall document the following metric: 23/2

- An upper bound on the lateness of the execution of a handler. That is, the maximum time between the time specified for the event and when a handler is actually invoked assuming no other handler or task is executing during this interval. 24/3

Implementation Advice

The protected handler procedure should be executed directly by the real-time clock interrupt mechanism. 25/2

NOTE 1 Since a call of Set_Handler is not a potentially blocking operation, it can be called from within a handler. 26/2

NOTE 2 A Timing_Event_Handler can be associated with several Timing_Event objects. 27/2

D.16 Multiprocessor Implementation

1/3 This subclause allows implementations on multiprocessor platforms to be configured.

Static Semantics

2/3 The following language-defined library package exists:

3/5
```
package System.Multiprocessors
   with Preelaborate, Nonblocking, Global => in out synchronized is
```

4/3
```
   type CPU_Range is range 0 .. implementation-defined;
   Not_A_Specific_CPU : constant CPU_Range := 0;
   subtype CPU is CPU_Range range 1 .. CPU_Range'Last;
```

5/3
```
   function Number_Of_CPUs return CPU;
end System.Multiprocessors;
```

6/3 A call of Number_Of_CPUs returns the number of processors available to the program. Within a given partition, each call on Number_Of_CPUs will return the same value.

7/5 For a task type (including the anonymous type of a single_task_declaration), protected type (including the anonymous type of a single_protected_declaration), or subprogram, the following language-defined representation aspect may be specified:

8/3 CPU The aspect CPU is an expression, which shall be of type System.Multiprocessors.CPU_Range.

Legality Rules

9/3 If the CPU aspect is specified for a subprogram, the expression shall be static.

10/5 The CPU aspect shall not be specified on a task or protected interface type.

10.1/6 An internal call shall not occur within the expression of a CPU aspect of a protected type.

Dynamic Semantics

11/5 The expression specified for the CPU aspect of a task or protected type is evaluated each time an object of the corresponding type is created (see 9.1 and 9.4). The CPU value is then associated with the object.

12/3 The CPU aspect has no effect if it is specified for a subprogram other than the main subprogram; the CPU value is not associated with any task.

13/3 The CPU value is associated with the environment task if the CPU aspect is specified for the main subprogram. If the CPU aspect is not specified for the main subprogram it is implementation defined on which processor the environment task executes.

14/5 For a task, the CPU value determines the processor on which the task will activate and execute; the task is said to be assigned to that processor. If the CPU value is Not_A_Specific_CPU, then the task is not assigned to a processor. A task without a CPU aspect specified will activate and execute on the same processor as its activating task if the activating task is assigned a processor. If the CPU value is not in the range of System.Multiprocessors.CPU_Range or is greater than Number_Of_CPUs the task is defined to have failed, and it becomes a completed task (see 9.2).

15/5 For a protected type, the CPU value determines the processor on which calling tasks will execute; the protected object is said to be assigned to that processor. If the CPU value is Not_A_Specific_CPU, then the protected object is not assigned to a processor. A call to a protected object that is assigned to a

processor from a task that is not assigned a processor or is assigned a different processor raises Program_Error.

Implementation Advice

Starting a protected action on a protected object statically assigned to a processor should be implemented without busy-waiting. 16/5

D.16.1 Multiprocessor Dispatching Domains

This subclause allows implementations on multiprocessor platforms to be partitioned into distinct dispatching domains during program startup. 1/3

Static Semantics

The following language-defined library package exists: 2/3

```
with Ada.Real_Time;                                                                    3/5
with Ada.Task_Identification;
package System.Multiprocessors.Dispatching_Domains
   with Nonblocking, Global => in out synchronized is
   Dispatching_Domain_Error : exception;                                               4/3
   type Dispatching_Domain (<>) is limited private;                                    5/3
   System_Dispatching_Domain : constant Dispatching_Domain;                            6/3
   function Create (First : CPU; Last : CPU_Range) return                              7/4
Dispatching_Domain;
   function Get_First_CPU (Domain : Dispatching_Domain) return CPU;                    8/3
   function Get_Last_CPU  (Domain : Dispatching_Domain) return CPU_Range;              9/4
   type CPU_Set is array(CPU range <>) of Boolean;                                     9.1/4
   function Create (Set : CPU_Set) return Dispatching_Domain;                          9.2/4
   function Get_CPU_Set (Domain : Dispatching_Domain) return CPU_Set;                  9.3/4
   function Get_Dispatching_Domain                                                     10/3
      (T   : Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task)
         return Dispatching_Domain;
   procedure Assign_Task                                                               11/3
      (Domain : in out Dispatching_Domain;
       CPU    : in     CPU_Range := Not_A_Specific_CPU;
       T      : in     Ada.Task_Identification.Task_Id :=
                  Ada.Task_Identification.Current_Task);
   procedure Set_CPU                                                                   12/3
      (CPU : in CPU_Range;
       T   : in Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task);
   function Get_CPU                                                                    13/3
      (T   : Ada.Task_Identification.Task_Id :=
                Ada.Task_Identification.Current_Task)
         return CPU_Range;
   procedure Delay_Until_And_Set_CPU                                                   14/3
      (Delay_Until_Time : in Ada.Real_Time.Time; CPU : in CPU_Range);
private                                                                                15/3
   ... -- not specified by the language
end System.Multiprocessors.Dispatching_Domains;
```

A *dispatching domain* represents a set of processors on which a task may execute. Each processor is contained within exactly one dispatching domain. An object of type Dispatching_Domain identifies a 16/4

dispatching domain. System_Dispatching_Domain identifies a domain that contains the processor or processors on which the environment task executes. At program start-up all processors are contained within this domain.

17/3 For a task type (including the anonymous type of a single_task_declaration), the following language-defined representation aspect may be specified:

18/3 Dispatching_Domain

The value of aspect Dispatching_Domain is an expression, which shall be of type Dispatching_Domains.Dispatching_Domain. This aspect is the domain to which the task (or all objects of the task type) are assigned.

Legality Rules

19/3 The Dispatching_Domain aspect shall not be specified for a task interface.

Dynamic Semantics

20/4 The expression specified for the Dispatching_Domain aspect of a task type is evaluated each time an object of the task type is created (see 9.1). If the identified dispatching domain is empty, then Dispatching_Domain_Error is raised; otherwise the newly created task is assigned to the domain identified by the value of the expression.

21/3 If a task is not explicitly assigned to any domain, it is assigned to that of the activating task. A task always executes on some CPU in its domain.

22/4 If both the dispatching domain and CPU are specified for a task, and the CPU value is not contained within the set of processors for the domain (and is not Not_A_Specific_CPU), the activation of the task is defined to have failed, and it becomes a completed task (see 9.2).

23/4 The function Create with First and Last parameters creates and returns a dispatching domain containing all the processors in the range First .. Last. The function Create with a Set parameter creates and returns a dispatching domain containing the processors for which Set(I) is True. These processors are removed from System_Dispatching_Domain. A call of Create will raise Dispatching_Domain_Error if any designated processor is not currently in System_Dispatching_Domain, or if the system cannot support a distinct domain over the processors identified, or if a processor has a task assigned to it, or if the allocation would leave System_Dispatching_Domain empty. A call of Create will raise Dispatching_Domain_Error if the calling task is not the environment task, or if Create is called after the call to the main subprogram.

24/4 The function Get_First_CPU returns the first CPU in Domain, or CPU'First if Domain is empty; Get_Last_CPU returns the last CPU in Domain, or CPU_Range'First if Domain is empty. The function Get_CPU_Set(D) returns an array whose low bound is Get_First_CPU(D), whose high bound is Get_Last_CPU(D), with True values in the Set corresponding to the CPUs that are in the given Domain.

25/4 The function Get_Dispatching_Domain returns the dispatching domain on which the task is assigned.

26/4 A call of the procedure Assign_Task assigns task T to the CPU within the dispatching domain Domain. Task T can now execute only on CPU, unless CPU designates Not_A_Specific_CPU in which case it can execute on any processor within Domain. The exception Dispatching_Domain_Error is propagated if Domain is empty, T is already assigned to a dispatching domain other than System_Dispatching_Domain, or if CPU is not one of the processors of Domain (and is not Not_A_Specific_CPU). A call of Assign_Task is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical. Assigning a task already assigned to System_Dispatching_Domain to that domain has no effect.

A call of procedure Set_CPU assigns task T to the CPU. Task T can now execute only on CPU, unless CPU designates Not_A_Specific_CPU, in which case it can execute on any processor within its dispatching domain. The exception Dispatching_Domain_Error is propagated if CPU is not one of the processors of the dispatching domain on which T is assigned (and is not Not_A_Specific_CPU). A call of Set_CPU is a task dispatching point for task T unless T is inside of a protected action, in which case the effect on task T is delayed until its next task dispatching point. If T is the Current_Task the effect is immediate if T is not inside a protected action, otherwise the effect is as soon as practical. 27/4

The function Get_CPU returns the processor assigned to task T, or Not_A_Specific_CPU if the task is not assigned to a processor. 28/3

A call of Delay_Until_And_Set_CPU delays the calling task for the designated time and then assigns the task to the specified processor when the delay expires. The exception Dispatching_Domain_Error is propagated if P is not one of the processors of the calling task's dispatching domain (and is not Not_A_Specific_CPU). 29/4

Implementation Requirements

The implementation shall perform the operations Assign_Task, Set_CPU, Get_CPU and Delay_Until_And_Set_CPU atomically with respect to any of these operations on the same dispatching_domain, processor or task. 30/3

Any task that belongs to the system dispatching domain can execute on any CPU within that domain, unless the assignment of the task has been specified. 30.1/4

Implementation Advice

Each dispatching domain should have separate and disjoint ready queues. 31/3

Documentation Requirements

The implementation shall document the processor(s) on which the clock interrupt is handled and hence where delay queue and ready queue manipulations occur. For any Interrupt_Id whose handler can execute on more than one processor the implementation shall also document this set of processors. 32/3

Implementation Permissions

An implementation may limit the number of dispatching domains that can be created and raise Dispatching_Domain_Error if an attempt is made to exceed this number. 33/3

The implementation may defer the effect of a Set_CPU or an Assign_Task operation until the specified task leaves an ongoing parallel construct. 34/5

Annex E
(normative)
Distributed Systems

This Annex defines facilities for supporting the implementation of distributed systems using multiple partitions working cooperatively as part of a single Ada program. 1

Post-Compilation Rules

A *distributed system* is an interconnection of one or more *processing nodes* (a system resource that has both computational and storage capabilities), and zero or more *storage nodes* (a system resource that has only storage capabilities, with the storage addressable by one or more processing nodes). 2

A *distributed program* comprises one or more partitions that execute independently (except when they communicate) in a distributed system. 3

The process of mapping the partitions of a program to the nodes in a distributed system is called *configuring the partitions of the program*. 4

Implementation Requirements

The implementation shall provide means for explicitly assigning library units to a partition and for the configuring and execution of a program consisting of multiple partitions on a distributed system; the means are implementation defined. 5

Implementation Permissions

An implementation may require that the set of processing nodes of a distributed system be homogeneous. 6

NOTE 1 The partitions comprising a program can be executed on differently configured distributed systems or on a nondistributed system without requiring recompilation. A distributed program can be partitioned differently from the same set of library units without recompilation. The resulting execution is semantically equivalent. 7/5

NOTE 2 A distributed program retains the same type safety as the equivalent single partition program. 8

E.1 Partitions

The partitions of a distributed program are classified as either active or passive. 1

Post-Compilation Rules

An *active partition* is a partition as defined in 10.2. A *passive partition* is a partition that has no thread of control of its own, whose library units are all preelaborated, and whose data and subprograms are accessible to one or more active partitions. 2

A passive partition shall include only library_items that either are declared pure or are shared passive (see 10.2.1 and E.2.1). 3

An active partition shall be configured on a processing node. A passive partition shall be configured either on a storage node or on a processing node. 4

The configuration of the partitions of a program onto a distributed system shall be consistent with the possibility for data references or calls between the partitions implied by their semantic dependences. Any reference to data or call of a subprogram across partitions is called a *remote access*. 5

Dynamic Semantics

6 A library_item is elaborated as part of the elaboration of each partition that includes it. If a normal library unit (see E.2) has state, then a separate copy of the state exists in each active partition that elaborates it. The state evolves independently in each such partition.

7 An active partition *terminates* when its environment task terminates. A partition becomes *inaccessible* if it terminates or if it is *aborted*. An active partition is aborted when its environment task is aborted. In addition, if a partition fails during its elaboration, it becomes inaccessible to other partitions. Other implementation-defined events can also result in a partition becoming inaccessible.

8/1 For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit, the following attribute is defined:

9 D'Partition_Id

Denotes a value of the type *universal_integer* that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see E.2.3) the given partition is the one where the body of D was elaborated.

Bounded (Run-Time) Errors

10/2 It is a bounded error for there to be cyclic elaboration dependences between the active partitions of a single distributed program. The possible effects, in each of the partitions involved, are deadlock during elaboration, or the raising of Communication_Error or Program_Error.

Implementation Permissions

11 An implementation may allow multiple active or passive partitions to be configured on a single processing node, and multiple passive partitions to be configured on a single storage node. In these cases, the scheduling policies, treatment of priorities, and management of shared resources between these partitions are implementation defined.

12 An implementation may allow separate copies of an active partition to be configured on different processing nodes, and to provide appropriate interactions between the copies to present a consistent state of the partition to other active partitions.

13/5 In an implementation, the partitions of a distributed program may be loaded and elaborated at different times; they may be loaded and elaborated one at a time over an extended period of time. An implementation may provide facilities to abort and reload a partition during the execution of a distributed program.

14 An implementation may allow the state of some of the partitions of a distributed program to persist while other partitions of the program terminate and are later reinvoked.

15/5 NOTE 1 Library units are grouped into partitions after compile time, but before run time. At compile time, only the relevant library unit properties are identified using categorization aspects.

16 NOTE 2 The value returned by the Partition_Id attribute can be used as a parameter to implementation-provided subprograms in order to query information about the partition.

E.2 Categorization of Library Units

1 Library units can be categorized according to the role they play in a distributed program. Certain restrictions are associated with each category to ensure that the semantics of a distributed program remain close to the semantics for a nondistributed program.

A *categorization aspect* restricts the declarations, child units, or semantic dependences of the library unit to which it applies. A *categorized library unit* is a library unit that has a categorization aspect that is True. 2/3

The aspects Shared_Passive, Remote_Types, and Remote_Call_Interface are categorization aspects. In addition, for the purposes of this Annex, the aspect Pure (see 10.2.1) is considered a categorization aspect. 3/3

A library package or generic library package is called a *shared passive* library unit if the Shared_Passive aspect of the unit is True. A library package or generic library package is called a *remote types* library unit if the Remote_Types aspect of the unit is True. A library unit is called a *remote call interface* if the Remote_Call_Interface aspect of the unit is True. A *normal library unit* is one for which no categorization aspect is True. 4/3

The various categories of library units and the associated restrictions are described in this and the following subclauses. The categories are related hierarchically in that the library units of one category can depend semantically only on library units of that category or an earlier one in the hierarchy, except that the body of a remote types or remote call interface library unit is unrestricted, the declaration of a remote types or remote call interface library unit may depend on preelaborated normal library units that are mentioned only in private with clauses, and all categories can depend on limited views. 5/3

The overall hierarchy (including declared pure) is as follows, with a lower-numbered category being "earlier in the hierarchy" in the sense of the previous paragraph: 6/3

1. Declared Pure 6.1/3
2. Shared Passive 6.2/3
3. Remote Types 6.3/3
4. Remote Call Interface 6.4/3
5. Normal (no restrictions) 6.5/3

Paragraphs 7 through 11 were deleted.

Declared pure and shared passive library units are preelaborated. The declaration of a remote types or remote call interface library unit is required to be preelaborable. 12

Paragraph 13 and 14 were deleted.

E.2.1 Shared Passive Library Units

A shared passive library unit is used for managing global data shared between active partitions. The restrictions on shared passive library units prevent the data or tasks of one active partition from being accessible to another active partition through references implicit in objects declared in the shared passive library unit. 1

Paragraphs 2 and 3 were moved to Annex J, "Obsolescent Features".

Legality Rules

When the library unit aspect (see 13.1.1) Shared_Passive of a library unit is True, the library unit is a *shared passive library unit*. The following restrictions apply to such a library unit: 4/5

- it shall be preelaborable (see 10.2.1); 5
- it shall depend semantically only upon declared pure or shared passive library_items; 6/3
- it shall not contain a library-level declaration: 7/5

7.1/5 • of an access type that designates a class-wide type;

7.2/5 • of a type with a part that is of a task type;

7.3/5 • of a type with a part that is of a protected type with entry_declarations; or

7.4/5 • that contains a name that denotes a type declared within a declared-pure package, if that type has a part that is of an access type; for the purposes of this rule, the parts considered include those of the full views of any private types or private extensions.

8 Notwithstanding the definition of accessibility given in 3.10.2, the declaration of a library unit P1 is not accessible from within the declarative region of a shared passive library unit P2, unless the shared passive library unit P2 depends semantically on P1.

Static Semantics

9 A shared passive library unit is preelaborated.

Post-Compilation Rules

10 A shared passive library unit shall be assigned to at most one partition within a given program.

11 Notwithstanding the rule given in 10.2, a compilation unit in a given partition does not *need* (in the sense of 10.2) the shared passive library units on which it depends semantically to be included in that same partition; they will typically reside in separate passive partitions.

E.2.2 Remote Types Library Units

1 A remote types library unit supports the definition of types intended for use in communication between active partitions.

Paragraphs 2 and 3 were moved to Annex J, "Obsolescent Features".

Legality Rules

4/5 When the library unit aspect (see 13.1.1) Remote_Types of a library unit is True, the library unit is a *remote types library unit*. The following restrictions apply to such a library unit:

5 • it shall be preelaborable;

6/3 • it shall depend semantically only on declared pure library_items, shared passive library units, other remote types library units, or preelaborated normal library units that are mentioned only in private with clauses;

7 • it shall not contain the declaration of any variable within the visible part of the library unit;

8/2 • the full view of each type declared in the visible part of the library unit that has any available stream attributes shall support external streaming (see 13.13.2).

9/3 A named access type declared in the visible part of a remote types or remote call interface library unit is called a *remote access type*. Such a type shall be:

9.1/1 • an access-to-subprogram type, or

9.2/3 • a general access type that designates a class-wide limited private type, a class-wide limited interface type, or a class-wide private extension all of whose ancestors are either private extensions, limited interface types, or limited private types.

9.3/1 A type that is derived from a remote access type is also a remote access type.

9.4/5 A remote access-to-subprogram type shall not be nonblocking (see 9.5).

The following restrictions apply to the use of a remote access-to-subprogram type: 10

- A value of a remote access-to-subprogram type shall be converted only to or from another (subtype-conformant) remote access-to-subprogram type; 11/2
- The prefix of an Access attribute_reference that yields a value of a remote access-to-subprogram type shall statically denote a (subtype-conformant) remote subprogram. 12

The following restrictions apply to the use of a remote access-to-class-wide type: 13

- The primitive subprograms of the corresponding specific type shall only have access parameters if they are controlling formal parameters. The primitive functions of the corresponding specific type shall only have an access result if it is a controlling access result. Each noncontrolling formal parameter and noncontrolling result type shall support external streaming (see 13.13.2); 14/3
- The corresponding specific type shall not have a primitive procedure with the Synchronization aspect specified unless the synchronization_kind is Optional (see 9.5); 14.1/3
- A value of a remote access-to-class-wide type shall be explicitly converted only to another remote access-to-class-wide type; 15
- A value of a remote access-to-class-wide type shall be dereferenced (or implicitly converted to an anonymous access type) only as part of a dispatching call to a primitive operation of the designated type where the value designates a controlling operand of the call (see E.4, “Remote Subprogram Calls”); 16/4
- A controlling access result value for a primitive function with any controlling operands of the corresponding specific type shall either be explicitly converted to a remote access-to-class-wide type or be part of a dispatching call where the value designates a controlling operand of the call; 16.1/3
- The Storage_Pool attribute is not defined for a remote access-to-class-wide type; the expected type for an allocator shall not be a remote access-to-class-wide type. A remote access-to-class-wide type shall not be an actual parameter for a generic formal access type. The Storage_Size attribute of a remote access-to-class-wide type yields 0. The Storage_Pool and Storage_Size aspects shall not be specified for a remote access-to-class-wide type. 17/4

Erroneous Execution

Execution is erroneous if some operation (other than the initialization or finalization of the object) modifies the value of a constant object declared in the visible part of a remote types package. 17.1/4

NOTE 1 A remote types library unit is not necessarily pure, and the types it defines can include levels of indirection implemented by using access types. User-specified Read and Write attributes (see 13.13.2) provide for sending values of such a type between active partitions, with Write marshalling the representation, and Read unmarshalling any levels of indirection. 18/5

NOTE 2 The value of a remote access-to-class-wide limited interface can designate an object of a nonlimited type derived from the interface. 19/3

NOTE 3 A remote access type can designate a class-wide synchronized, protected, or task interface type. 20/5

E.2.3 Remote Call Interface Library Units

A remote call interface library unit can be used as an interface for remote procedure calls (RPCs) (or remote function calls) between active partitions. 1

Paragraphs 2 through 6 were moved to Annex J, “Obsolescent Features”.

Legality Rules

7/5 When the library unit aspect (see 13.1.1) Remote_Call_Interface of a library unit is True, the library unit is a *remote call interface (RCI)*. A subprogram declared in the visible part of such a library unit, or declared by such a library unit, is called a *remote subprogram*.

8/3 The declaration of an RCI library unit shall be preelaborable (see 10.2.1), and shall depend semantically only upon declared pure library_items, shared passive library units, remote types library units, other remote call interface library units, or preelaborated normal library units that are mentioned only in private with clauses.

9/1 In addition, the following restrictions apply to an RCI library unit:

10/1 • its visible part shall not contain the declaration of a variable;

11/1 • its visible part shall not contain the declaration of a limited type;

12/1 • its visible part shall not contain a nested generic_declaration;

13/3 • it shall not be, nor shall its visible part contain, the declaration of a subprogram for which aspect Inline is True;

13.1/5 • it shall not be, nor shall its visible part contain, the declaration of a subprogram that is nonblocking (see 9.5);

14/3 • it shall not be, nor shall its visible part contain, a subprogram (or access-to-subprogram) declaration whose profile has a parameter or result of a type that does not support external streaming (see 13.13.2);

15 • any public child of the library unit shall be a remote call interface library unit.

15.1/5 Specification of a stream-oriented attribute is illegal in the specification of a remote call interface library unit. In addition to the places where Legality Rules normally apply (see 12.3), this rule applies also in the private part of an instance of a generic unit.

16/5 All_Calls_Remote is a library unit aspect. If the All_Calls_Remote aspect of a library unit is True, the library unit shall be a remote call interface.

Post-Compilation Rules

17 A remote call interface library unit shall be assigned to at most one partition of a given program. A remote call interface library unit whose parent is also an RCI library unit shall be assigned only to the same partition as its parent.

18 Notwithstanding the rule given in 10.2, a compilation unit in a given partition that semantically depends on the declaration of an RCI library unit, *needs* (in the sense of 10.2) only the declaration of the RCI library unit, not the body, to be included in that same partition. Therefore, the body of an RCI library unit is included only in the partition to which the RCI library unit is explicitly assigned.

Implementation Requirements

19/4 If aspect All_Calls_Remote is True for a given RCI library unit, then the implementation shall route any of the following calls through the Partition Communication Subsystem (PCS); see E.5:

19.1/4 • A direct call to a subprogram of the RCI unit from outside the declarative region of the unit;

19.2/4 • An indirect call through a remote access-to-subprogram value that designates a subprogram of the RCI unit;

19.3/4 • A dispatching call with a controlling operand designated by a remote access-to-class-wide value whose tag identifies a type declared in the RCI unit.

Implementation Permissions

An implementation may omit support for the Remote_Call_Interface aspect or the All_Calls_Remote aspect. Explicit message-based communication between active partitions can be supported as an alternative to RPC. 20/5

E.3 Consistency of a Distributed System

This subclause defines attributes and rules associated with verifying the consistency of a distributed program. 1/3

Static Semantics

For a prefix P that statically denotes a program unit, the following attributes are defined: 2/1

P'Version Yields a value of the predefined type String that identifies the version of the compilation unit that contains the declaration of the program unit. 3

P'Body_Version 4

Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit.

The *version* of a compilation unit changes whenever the compilation unit changes in a semantically significant way. This document does not define the exact meaning of "semantically significant". It is unspecified whether there are other events (such as recompilation) that result in the version of a compilation unit changing. 5/1

If P is not a library unit, and P has no completion, then P'Body_Version returns the Body_Version of the innermost program unit enclosing the declaration of P. If P is a library unit, and P has no completion, then P'Body_Version returns a value that is different from Body_Version of any version of P that has a completion. 5.1/1

Bounded (Run-Time) Errors

In a distributed program, a library unit is *consistent* if the same version of its declaration is used throughout. It is a bounded error to elaborate a partition of a distributed program that contains a compilation unit that depends on a different version of the declaration of a shared passive or RCI library unit than that included in the partition to which the shared passive or RCI library unit was assigned. As a result of this error, Program_Error can be raised in one or both partitions during elaboration; in any case, the partitions become inaccessible to one another. 6

E.4 Remote Subprogram Calls

A *remote subprogram call* is a subprogram call that invokes the execution of a subprogram in another (active) partition. The partition that originates the remote subprogram call is the *calling partition*, and the partition that executes the corresponding subprogram body is the *called partition*. Some remote procedure calls are allowed to return prior to the completion of subprogram execution. These are called *asynchronous remote procedure calls*. 1/5

There are three different ways of performing a remote subprogram call: 2

- As a direct call on a (remote) subprogram explicitly declared in a remote call interface; 3
- As an indirect call through a value of a remote access-to-subprogram type; 4

- As a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type.

The first way of calling corresponds to a *static* binding between the calling and the called partition. The latter two ways correspond to a *dynamic* binding between the calling and the called partition.

Remote types library units (see E.2.2) and remote call interface library units (see E.2.3) define the remote subprograms or remote access types used for remote subprogram calls.

Legality Rules

In a dispatching call with two or more controlling operands, if one controlling operand is designated by a value of a remote access-to-class-wide type, then all shall be.

A nonblocking program unit shall not contain, other than within nested units with Nonblocking specified as statically False, a dispatching call with a controlling operand designated by a value of a remote access-to-class-wide type.

Dynamic Semantics

For the execution of a remote subprogram call, subprogram parameters (and later the results, if any) are passed using a stream-oriented representation (see 13.13.1) which is suitable for transmission between partitions. This action is called *marshalling*. *Unmarshalling* is the reverse action of reconstructing the parameters or results from the stream-oriented representation. Marshalling is performed initially as part of the remote subprogram call in the calling partition; unmarshalling is done in the called partition. After the remote subprogram completes, marshalling is performed in the called partition, and finally unmarshalling is done in the calling partition.

A *calling stub* is the sequence of code that replaces the subprogram body of a remotely called subprogram in the calling partition. A *receiving stub* is the sequence of code (the “wrapper”) that receives a remote subprogram call on the called partition and invokes the appropriate subprogram body.

Remote subprogram calls are executed at most once, that is, if the subprogram call returns normally, then the called subprogram's body was executed exactly once.

The task executing a remote subprogram call blocks until the subprogram in the called partition returns, unless the call is asynchronous. For an asynchronous remote procedure call, the calling task can become ready before the procedure in the called partition returns.

If a construct containing a remote call is aborted, the remote subprogram call is *cancelled*. Whether the execution of the remote subprogram is immediately aborted as a result of the cancellation is implementation defined.

If a remote subprogram call is received by a called partition before the partition has completed its elaboration, the call is kept pending until the called partition completes its elaboration (unless the call is cancelled by the calling partition prior to that).

If an exception is propagated by a remotely called subprogram, and the call is not an asynchronous call, the corresponding exception is reraised at the point of the remote subprogram call. For an asynchronous call, if the remote procedure call returns prior to the completion of the remotely called subprogram, any exception is lost.

The exception Communication_Error (see E.5) is raised if a remote call cannot be completed due to difficulties in communicating with the called partition.

All forms of remote subprogram calls are potentially blocking operations (see 9.5).

In a remote subprogram call with a formal parameter of a class-wide type, a check is made that the tag of the actual parameter identifies a tagged type declared in a declared-pure or shared passive library unit, or in the visible part of a remote types or remote call interface library unit. Program_Error is raised if this check fails. In a remote function call which returns a class-wide type, the same check is made on the function result. 18/1

In a dispatching call with two or more controlling operands that are designated by values of a remote access-to-class-wide type, a check is made (in addition to the normal Tag_Check — see 11.5) that all the remote access-to-class-wide values originated from Access attribute_references that were evaluated by tasks of the same active partition. Constraint_Error is raised if this check fails. 19

Implementation Requirements

The implementation of remote subprogram calls shall conform to the PCS interface as defined by the specification of the language-defined package System.RPC (see E.5). The calling stub shall use the Do_RPC procedure unless the remote procedure call is asynchronous in which case Do_APC shall be used. On the receiving side, the corresponding receiving stub shall be invoked by the RPC-receiver. 20

With respect to shared variables in shared passive library units, the execution of the corresponding subprogram body of a synchronous remote procedure call is considered to be part of the execution of the calling task. The execution of the corresponding subprogram body of an asynchronous remote procedure call proceeds in parallel with the calling task and does not signal the next action of the calling task (see 9.10). 20.1/1

NOTE 1 A given active partition can both make and receive remote subprogram calls. Thus, an active partition can act as both a client and a server. 21

NOTE 2 If a given exception is propagated by a remote subprogram call, but the exception does not exist in the calling partition, the exception can be handled by an **others** choice or be propagated to and handled by a third partition. 22

E.4.1 Asynchronous Remote Calls

This subclause introduces the aspect Asynchronous which can be specified to allow a remote subprogram call to return prior to completion of the execution of the corresponding remote subprogram body. 1/3

Paragraphs 2 through 7 were deleted.

Static Semantics

For a remote procedure, the following language-defined representation aspect may be specified: 8/3

Asynchronous The type of aspect Asynchronous is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified, the aspect is False. 8.1/3

For a remote access type, the following language-defined representation aspect may be specified: 8.2/3

Asynchronous The type of aspect Asynchronous is Boolean. If directly specified, the aspect_definition shall be a static expression. If not specified (including by inheritance), the aspect is False. 8.3/3

Legality Rules

If aspect Asynchronous is specified for a remote procedure, the formal parameters of the procedure shall all be of mode **in**. 8.4/3

If aspect Asynchronous is specified for a remote access type, the type shall be a remote access-to-class-wide type, or the type shall be a remote access-to-procedure type with the formal parameters of the designated profile of the type all of mode **in**. 8.5/3

Dynamic Semantics

9/3 A remote call is *asynchronous* if it is a call to a procedure, or a call through a value of an access-to-procedure type, for which aspect Asynchronous is True. In addition, if aspect Asynchronous is True for a remote access-to-class-wide type, then a dispatching call on a procedure with a controlling operand designated by a value of the type is asynchronous if the formal parameters of the procedure are all of mode **in**.

Implementation Requirements

10 Asynchronous remote procedure calls shall be implemented such that the corresponding body executes at most once as a result of the call.

E.4.2 Example of Use of a Remote Access-to-Class-Wide Type

Examples

1 *Example of using a remote access-to-class-wide type to achieve dynamic binding across active partitions:*

2/5
```
package Tapes
   with Pure is
   type Tape is abstract tagged limited private;
   -- Primitive dispatching operations where
   -- Tape is controlling operand
   procedure Copy (From, To : access Tape;
                   Num_Recs : in Natural) is abstract;
   procedure Rewind (T : access Tape) is abstract;
   -- More operations
private
   type Tape is ...
end Tapes;
```

3/5
```
with Tapes;
package Name_Server
   with Remote_Call_Interface is
   -- Dynamic binding to remote operations is achieved
   -- using the access-to-limited-class-wide type Tape_Ptr
   type Tape_Ptr is access all Tapes.Tape'Class;
   -- The following statically bound remote operations
   -- allow for a name-server capability in this example
   function  Find     (Name : String) return Tape_Ptr;
   procedure Register (Name : in String; T : in Tape_Ptr);
   procedure Remove   (T : in Tape_Ptr);
   -- More operations
end Name_Server;
```

4
```
package Tape_Driver is
  -- Declarations are not shown, they are irrelevant here
end Tape_Driver;
```

5/5
```
with Tapes, Name_Server;
package body Tape_Driver is
   type New_Tape is new Tapes.Tape with ...
   overriding
   procedure Rewind (T : access New_Tape);
   overriding
   procedure Copy
    (From, To : access New_Tape; Num_Recs: in Natural) is
   begin
      . . .
   end Copy;
   procedure Rewind (T : access New_Tape) is
   begin
      . . .
   end Rewind;
   -- Objects remotely accessible through use
   -- of Name_Server operations
   Tape1, Tape2 : aliased New_Tape;
begin
   Name_Server.Register ("NINE-TRACK",  Tape1'Access);
   Name_Server.Register ("SEVEN-TRACK", Tape2'Access);
end Tape_Driver;
```

6
```
with Tapes, Name_Server;
-- Tape_Driver is not needed and thus not mentioned in the with_clause
procedure Tape_Client is
   T1, T2 : Name_Server.Tape_Ptr;
begin
   T1 := Name_Server.Find ("NINE-TRACK");
   T2 := Name_Server.Find ("SEVEN-TRACK");
   Tapes.Rewind (T1);
   Tapes.Rewind (T2);
   Tapes.Copy (T1, T2, 3);
end Tape_Client;
```

Discussion of the example: 7/5

This paragraph was deleted. 8/1

- The package Tapes provides the necessary declarations of the type and its primitive operations. 9
- Name_Server is a remote call interface package and is elaborated in a separate active partition to provide the necessary naming services (such as Register and Find) to the entire distributed program through remote subprogram calls. 10
- Tape_Driver is a normal package that is elaborated in a partition configured on the processing node that is connected to the tape device(s). The abstract operations are overridden to support the locally declared tape devices (Tape1, Tape2). The package is not visible to its clients, but it exports the tape devices (as remote objects) through the services of the Name_Server. This allows for tape devices to be dynamically added, removed or replaced without requiring the modification of the clients' code. 11
- The Tape_Client procedure references only declarations in the Tapes and Name_Server packages. Before using a tape for the first time, it will query the Name_Server for a system-wide identity for that tape. From then on, it can use that identity to access the tape device. 12/5
- Values of remote access type Tape_Ptr include the necessary information to complete the remote dispatching operations that result from dereferencing the controlling operands T1 and T2. 13

E.5 Partition Communication Subsystem

1/2 The *Partition Communication Subsystem* (PCS) provides facilities for supporting communication between the active partitions of a distributed program. The package System.RPC is a language-defined interface to the PCS.

Static Semantics

2 The following language-defined library package exists:

3/5
```
with Ada.Streams; -- see 13.13.1
package System.RPC
   with Nonblocking => False, Global => in out synchronized is
```

4
```
   type Partition_Id is range 0 .. implementation-defined;
```

5
```
   Communication_Error : exception;
```

6
```
   type Params_Stream_Type (
      Initial_Size : Ada.Streams.Stream_Element_Count) is new
      Ada.Streams.Root_Stream_Type with private;
```

7
```
   procedure Read(
      Stream : in out Params_Stream_Type;
      Item : out Ada.Streams.Stream_Element_Array;
      Last : out Ada.Streams.Stream_Element_Offset);
```

8
```
   procedure Write(
      Stream : in out Params_Stream_Type;
      Item : in Ada.Streams.Stream_Element_Array);
```

9
```
   -- Synchronous call
   procedure Do_RPC(
      Partition  : in Partition_Id;
      Params     : access Params_Stream_Type;
      Result     : access Params_Stream_Type);
```

10
```
   -- Asynchronous call
   procedure Do_APC(
      Partition  : in Partition_Id;
      Params     : access Params_Stream_Type);
```

11
```
   -- The handler for incoming RPCs
   type RPC_Receiver is access procedure(
      Params     : access Params_Stream_Type;
      Result     : access Params_Stream_Type);
```

12
```
   procedure Establish_RPC_Receiver(
      Partition : in Partition_Id;
      Receiver  : in RPC_Receiver);
```

13
```
private
   ... -- not specified by the language
end System.RPC;
```

14 A value of the type Partition_Id is used to identify a partition.

15 An object of the type Params_Stream_Type is used for identifying the particular remote subprogram that is being called, as well as marshalling and unmarshalling the parameters or result of a remote subprogram call, as part of sending them between partitions.

16 The Read and Write procedures override the corresponding abstract operations for the type Params_Stream_Type.

Dynamic Semantics

17 The Do_RPC and Do_APC procedures send a message to the active partition identified by the Partition parameter.

After sending the message, Do_RPC blocks the calling task until a reply message comes back from the called partition or some error is detected by the underlying communication system in which case Communication_Error is raised at the point of the call to Do_RPC. 18

Do_APC operates in the same way as Do_RPC except that it is allowed to return immediately after sending the message. 19

Upon normal return, the stream designated by the Result parameter of Do_RPC contains the reply message. 20

The procedure System.RPC.Establish_RPC_Receiver is called once, immediately after elaborating the library units of an active partition (that is, right after the *elaboration of the partition*) if the partition includes an RCI library unit, but prior to invoking the main subprogram, if any. The Partition parameter is the Partition_Id of the active partition being elaborated. The Receiver parameter designates an implementation-provided procedure called the *RPC-receiver* which will handle all RPCs received by the partition from the PCS. Establish_RPC_Receiver saves a reference to the RPC-receiver; when a message is received at the called partition, the RPC-receiver is called with the Params stream containing the message. When the RPC-receiver returns, the contents of the stream designated by Result is placed in a message and sent back to the calling partition. 21

If a call on Do_RPC is aborted, a cancellation message is sent to the called partition, to request that the execution of the remotely called subprogram be aborted. 22

This paragraph was deleted. 23/5

Implementation Requirements

The implementation of the RPC-receiver shall be reentrant, thereby allowing concurrent calls on it from the PCS to service concurrent remote subprogram calls into the partition. 24

An implementation shall not restrict the replacement of the body of System.RPC. An implementation shall not restrict children of System.RPC. The related implementation permissions in the introduction to Annex A do not apply. 24.1/1

If the implementation of System.RPC is provided by the user, an implementation shall support remote subprogram calls as specified. 24.2/1

Documentation Requirements

The implementation of the PCS shall document whether the RPC-receiver is invoked from concurrent tasks. If there is an upper limit on the number of such tasks, this limit shall be documented as well, together with the mechanisms to configure it (if this is supported). 25

Implementation Permissions

The PCS is allowed to contain implementation-defined interfaces for explicit message passing, broadcasting, etc. Similarly, it is allowed to provide additional interfaces to query the state of some remote partition (given its partition ID) or of the PCS itself, to set timeouts and retry parameters, to get more detailed error status, etc. These additional interfaces should be provided in child packages of System.RPC. 26

A body for the package System.RPC is not required to be supplied by the implementation. 27/5

An alternative declaration is allowed for package System.RPC as long as it provides a set of operations that is substantially equivalent to the specification defined in this subclause. 27.1/3

Implementation Advice

28 Whenever possible, the PCS on the called partition should allow for multiple tasks to call the RPC-receiver with different messages and should allow them to block until the corresponding subprogram body returns.

29 The Write operation on a stream of type Params_Stream_Type should raise Storage_Error if it runs out of space trying to write the Item into the stream.

30 NOTE The package System.RPC is not designed for direct calls by user programs. It is instead designed for use in the implementation of remote subprograms calls, being called by the calling stubs generated for a remote call interface library unit to initiate a remote call, and in turn calling back to an RPC-receiver that dispatches to the receiving stubs generated for the body of a remote call interface, to handle a remote call received from elsewhere.

Annex F
(normative)
Information Systems

This Annex provides a set of facilities relevant to Information Systems programming. These fall into several categories: 1

- an attribute definition clause specifying Machine_Radix for a decimal subtype; 2
- the package Decimal, which declares a set of constants defining the implementation's capacity for decimal types, and a generic procedure for decimal division; and 3
- the child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing, which support formatted and localized output of decimal data, based on “picture String” values. 4/2

Other relevant facilities can be found in: 3.5.9, “Fixed Point Types”; 3.5.10, “Operations of Fixed Point Types”; 4.6, “Type Conversions”; 13.3, “Operational and Representation Attributes”; A.10.9, “Input-Output for Real Types”; B.3, “Interfacing with C and C++”; B.4, “Interfacing with COBOL”; Annex G, “Numerics”. 5/5

The character and string handling packages in Annex A, “Predefined Language Environment” are also relevant for Information Systems. 6

Implementation Advice

If COBOL (respectively, C) is widely supported in the target environment, implementations supporting the Information Systems Annex should provide the child package Interfaces.COBOL (respectively, Interfaces.C) specified in Annex B and should support a *convention*_identifier of COBOL (respectively, C) for the Convention aspect (see Annex B), thus allowing Ada programs to interface with programs written in that language. 7/3

F.1 Machine_Radix Attribute Definition Clause

Static Semantics

The representation attribute Machine_Radix may be specified for a decimal first subtype (see 3.5.9) via an attribute_definition_clause; the expression of such a clause shall be static, and its value shall be 2 or 10. A value of 2 implies a binary base range; a value of 10 implies a decimal base range. 1/5

Implementation Advice

Packed decimal should be used as the internal representation for objects of subtype S when S'Machine_Radix = 10. 2

Examples

Example of Machine_Radix attribute definition clause: 3

```
type Money is delta 0.01 digits 15;
for Money'Machine_Radix use 10;
```
4

F.2 The Package Decimal

Static Semantics

1 The library package Decimal has the following declaration:

2/5
```
package Ada.Decimal
   with Pure is

3     Max_Scale : constant := implementation-defined;
      Min_Scale : constant := implementation-defined;

4     Min_Delta : constant := 10.0**(-Max_Scale);
      Max_Delta : constant := 10.0**(-Min_Scale);

5     Max_Decimal_Digits : constant := implementation-defined;

6/3   generic
         type Dividend_Type  is delta <> digits <>;
         type Divisor_Type   is delta <> digits <>;
         type Quotient_Type  is delta <> digits <>;
         type Remainder_Type is delta <> digits <>;
      procedure Divide (Dividend  : in Dividend_Type;
                        Divisor   : in Divisor_Type;
                        Quotient  : out Quotient_Type;
                        Remainder : out Remainder_Type)
         with Convention => Intrinsic;

7  end Ada.Decimal;
```

8 Max_Scale is the largest N such that 10.0**(–N) is allowed as a decimal type's delta. Its type is *universal_integer*.

9 Min_Scale is the smallest N such that 10.0**(–N) is allowed as a decimal type's delta. Its type is *universal_integer*.

10 Min_Delta is the smallest value allowed for *delta* in a decimal_fixed_point_definition. Its type is *universal_real*.

11 Max_Delta is the largest value allowed for *delta* in a decimal_fixed_point_definition. Its type is *universal_real*.

12 Max_Decimal_Digits is the largest value allowed for *digits* in a decimal_fixed_point_definition. Its type is *universal_integer*.

Static Semantics

13 The effect of Divide is as follows. The value of Quotient is Quotient_Type(Dividend/Divisor). The value of Remainder is Remainder_Type(Intermediate), where Intermediate is the difference between Dividend and the product of Divisor and Quotient; this result is computed exactly.

Implementation Requirements

14 Decimal.Max_Decimal_Digits shall be at least 18.

15 Decimal.Max_Scale shall be at least 18.

16 Decimal.Min_Scale shall be at most 0.

17 NOTE The effect of division yielding a quotient with control over rounding versus truncation is obtained by applying either the function attribute Quotient_Type'Round or the conversion Quotient_Type to the expression Dividend/Divisor.

F.3 Edited Output for Decimal Types

The child packages Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing provide localizable formatted text output, known as *edited output*, for decimal types. An edited output string is a function of a numeric value, program-specifiable locale elements, and a format control value. The numeric value is of some decimal type. The locale elements are: 1/2

- the currency string; 2
- the digits group separator character; 3
- the radix mark character; and 4
- the fill character that replaces leading zeros of the numeric value. 5

For Text_IO.Editing the edited output and currency strings are of type String, and the locale characters are of type Character. For Wide_Text_IO.Editing their types are Wide_String and Wide_Character, respectively. For Wide_Wide_Text_IO.Editing their types are Wide_Wide_String and Wide_Wide_-Character, respectively. 6/2

Each of the locale elements has a default value that can be replaced or explicitly overridden. 7

A format-control value is of the private type Picture; it determines the composition of the edited output string and controls the form and placement of the sign, the position of the locale elements and the decimal digits, the presence or absence of a radix mark, suppression of leading zeros, and insertion of particular character values. 8

A Picture object is composed from a String value, known as a *picture String*, that serves as a template for the edited output string, and a Boolean value that controls whether a string of all space characters is produced when the number's value is zero. A picture String comprises a sequence of one- or two-Character symbols, each serving as a placeholder for a character or string at a corresponding position in the edited output string. The picture String symbols fall into several categories based on their effect on the edited output string: 9

Decimal Digit:	'9'					10
Radix Control:	'.'	'V'				
Sign Control:	'+'	'–'	'<'	'>'	"CR"	"DB"
Currency Control:	'$'	'#'				
Zero Suppression:	'Z'	'*'				
Simple Insertion:	'_'	'B'	'0'	'/'		

The entries are not case-sensitive. Mixed- or lower-case forms for "CR" and "DB", and lower-case forms for 'V', 'Z', and 'B', have the same effect as the upper-case symbols shown. 11

An occurrence of a '9' Character in the picture String represents a decimal digit position in the edited output string. 12

A radix control Character in the picture String indicates the position of the radix mark in the edited output string: an actual character position for '.', or an assumed position for 'V'. 13

A sign control Character in the picture String affects the form of the sign in the edited output string. The '<' and '>' Character values indicate parentheses for negative values. A Character '+', '–', or '<' appears either singly, signifying a fixed-position sign in the edited output, or repeated, signifying a floating-position sign that is preceded by zero or more space characters and that replaces a leading 0. 14

15 A currency control Character in the picture String indicates an occurrence of the currency string in the edited output string. The '$' Character represents the complete currency string; the '#' Character represents one character of the currency string. A '$' Character appears either singly, indicating a fixed-position currency string in the edited output, or repeated, indicating a floating-position currency string that occurs in place of a leading 0. A sequence of '#' Character values indicates either a fixed- or floating-position currency string, depending on context.

16 A zero suppression Character in the picture String allows a leading zero to be replaced by either the space character (for 'Z') or the fill character (for '*').

17 A simple insertion Character in the picture String represents, in general, either itself (if '/' or '0'), the space character (if 'B'), or the digits group separator character (if '_'). In some contexts it is treated as part of a floating sign, floating currency, or zero suppression string.

18/2 An example of a picture String is "<###Z_ZZ9.99>". If the currency string is "kr", the separator character is ',', and the radix mark is '.' then the edited output string values for the decimal values 32.10 and –5432.10 are "bbkrbbb32.10b" and "(bkr5,432.10)", respectively, where 'b' indicates the space character.

19/2 The generic packages Text_IO.Decimal_IO, Wide_Text_IO.Decimal_IO, and Wide_Wide_Text_IO.Decimal_IO (see A.10.9, “Input-Output for Real Types”) provide text input and nonedited text output for decimal types.

20/2 NOTE A picture String is of type Standard.String, for all of Text_IO.Editing, Wide_Text_IO.Editing, and Wide_Wide_Text_IO.Editing.

F.3.1 Picture String Formation

1/3 A *well-formed picture String*, or simply *picture String*, is a String value that conforms to the syntactic rules, composition constraints, and character replication conventions specified in this subclause.

Dynamic Semantics

2/1 *This paragraph was deleted.*

3
```
picture_string ::=
  fixed_$_picture_string
| fixed_#_picture_string
| floating_currency_picture_string
| non_currency_picture_string
```

4
```
fixed_$_picture_string ::=
  [fixed_LHS_sign] fixed_$_char {direct_insertion} [zero_suppression]
    number [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] [zero_suppression]
    number fixed_$_char {direct_insertion} [RHS_sign]

| floating_LHS_sign number fixed_$_char {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] fixed_$_char {direct_insertion}
    all_zero_suppression_number {direct_insertion}  [RHS_sign]
```

| [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
fixed_$_char {direct_insertion} [RHS_sign]

| all_sign_number {direct_insertion} fixed_$_char {direct_insertion} [RHS_sign]

fixed_#_picture_string ::= 5
[fixed_LHS_sign] single_#_currency {direct_insertion}
[zero_suppression] number [RHS_sign]

| [fixed_LHS_sign] multiple_#_currency {direct_insertion}
zero_suppression number [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] [zero_suppression]
number fixed_#_currency {direct_insertion} [RHS_sign]

| floating_LHS_sign number fixed_#_currency {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] single_#_currency {direct_insertion}
all_zero_suppression_number {direct_insertion} [RHS_sign]

| [fixed_LHS_sign] multiple_#_currency {direct_insertion}
all_zero_suppression_number {direct_insertion} [RHS_sign]

| [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
fixed_#_currency {direct_insertion} [RHS_sign]

| all_sign_number {direct_insertion} fixed_#_currency {direct_insertion} [RHS_sign]

floating_currency_picture_string ::= 6
[fixed_LHS_sign] {direct_insertion} floating_$_currency number [RHS_sign]
| [fixed_LHS_sign] {direct_insertion} floating_#_currency number [RHS_sign]
| [fixed_LHS_sign] {direct_insertion} all_currency_number {direct_insertion} [RHS_sign]

non_currency_picture_string ::= 7
[fixed_LHS_sign {direct_insertion}] zero_suppression number [RHS_sign]
| [floating_LHS_sign] number [RHS_sign]
| [fixed_LHS_sign {direct_insertion}] all_zero_suppression_number {direct_insertion}
[RHS_sign]
| all_sign_number {direct_insertion}
| fixed_LHS_sign direct_insertion {direct_insertion} number [RHS_sign]

fixed_LHS_sign ::= LHS_Sign 8

LHS_Sign ::= + | – | < 9

fixed_$_char ::= $ 10

11 direct_insertion ::= simple_insertion

12 simple_insertion ::= _ | B | 0 | /

13 zero_suppression ::= Z {Z | context_sensitive_insertion} | fill_string

14 context_sensitive_insertion ::= simple_insertion

15 fill_string ::= * {* | context_sensitive_insertion}

16 number ::=
fore_digits [radix [aft_digits] {direct_insertion}]
| radix aft_digits {direct_insertion}

17 fore_digits ::= 9 {9 | direct_insertion}

18 aft_digits ::= {9 | direct_insertion} 9

19 radix ::= . | V

20 RHS_sign ::= + | – | > | CR | DB

21 floating_LHS_sign ::=
LHS_Sign {context_sensitive_insertion} LHS_Sign {LHS_Sign | context_sensitive_insertion}

22 single_#_currency ::= #

23 multiple_#_currency ::= ## {#}

24 fixed_#_currency ::= single_#_currency | multiple_#_currency

25 floating_$_currency ::=
$ {context_sensitive_insertion} $ {$ | context_sensitive_insertion}

26 floating_#_currency ::=
{context_sensitive_insertion} # {# | context_sensitive_insertion}

27 all_sign_number ::= all_sign_fore [radix [all_sign_aft]] [>]

28 all_sign_fore ::=
sign_char {context_sensitive_insertion} sign_char {sign_char | context_sensitive_insertion}

29 all_sign_aft ::= {all_sign_aft_char} sign_char

all_sign_aft_char ::= sign_char | context_sensitive_insertion

sign_char ::= + | – | <

all_currency_number ::= all_currency_fore [radix [all_currency_aft]]

all_currency_fore ::=
  currency_char {context_sensitive_insertion}
    currency_char {currency_char | context_sensitive_insertion}

all_currency_aft ::= {all_currency_aft_char} currency_char

all_currency_aft_char ::= currency_char | context_sensitive_insertion

currency_char ::= $ | #

all_zero_suppression_number ::= all_zero_suppression_fore [radix [all_zero_suppression_aft]]

all_zero_suppression_fore ::=
  zero_suppression_char {zero_suppression_char | context_sensitive_insertion}

all_zero_suppression_aft ::= {all_zero_suppression_aft_char} zero_suppression_char

all_zero_suppression_aft_char ::= zero_suppression_char | context_sensitive_insertion

zero_suppression_char ::= Z | *

The following composition constraints apply to a picture String:

- A floating_LHS_sign does not have occurrences of different LHS_Sign Character values.
- If a picture String has '<' as fixed_LHS_sign, then it has '>' as RHS_sign.
- If a picture String has '<' in a floating_LHS_sign or in an all_sign_number, then it has an occurrence of '>'.
- If a picture String has '+' or '–' as fixed_LHS_sign, in a floating_LHS_sign, or in an all_sign_number, then it has no RHS_sign or '>' character.
- An instance of all_sign_number does not have occurrences of different sign_char Character values.
- An instance of all_currency_number does not have occurrences of different currency_char Character values.
- An instance of all_zero_suppression_number does not have occurrences of different zero_-suppression_char Character values, except for possible case differences between 'Z' and 'z'.

A *replicable Character* is a Character that, by the above rules, can occur in two consecutive positions in a picture String.

A *Character replication* is a String

char `&` `'('` `&` *spaces* `&` *count_string* `&` `')'`

where *char* is a replicable Character, *spaces* is a String (possibly empty) comprising only space Character values, and *count_string* is a String of one or more decimal digit Character values. A Character replication in a picture String has the same effect as (and is said to be *equivalent to*) a String comprising *n* consecutive occurrences of *char*, where *n*=Integer'Value(*count_string*).

51 An *expanded picture String* is a picture String containing no Character replications.

52 NOTE Although a sign to the left of the number can float, a sign to the right of the number is in a fixed position.

F.3.2 Edited Output Generation

Dynamic Semantics

1 The contents of an edited output string are based on:

2 • A value, Item, of some decimal type Num,

3 • An expanded picture String Pic_String,

4 • A Boolean value, Blank_When_Zero,

5 • A Currency string,

6 • A Fill character,

7 • A Separator character, and

8 • A Radix_Mark character.

9 The combination of a True value for Blank_When_Zero and a '*' character in Pic_String is inconsistent; no edited output string is defined.

10 A layout error is identified in the rules below if leading nonzero digits of Item, character values of the Currency string, or a negative sign would be truncated; in such cases no edited output string is defined.

11 The edited output string has lower bound 1 and upper bound N where N = Pic_String'Length + Currency_Length_Adjustment – Radix_Adjustment, and

12 • Currency_Length_Adjustment = Currency'Length – 1 if there is some occurrence of '$' in Pic_String, and 0 otherwise.

13 • Radix_Adjustment = 1 if there is an occurrence of 'V' or 'v' in Pic_Str, and 0 otherwise.

14 Let the magnitude of Item be expressed as a base-10 number $I_p \cdots I_1.F_1 \cdots F_q$, called the *displayed magnitude* of Item, where:

15 • q = Min(Max(Num'Scale, 0), n) where n is 0 if Pic_String has no radix and is otherwise the number of digit positions following radix in Pic_String, where a digit position corresponds to an occurrence of '9', a zero_suppression_char (for an all_zero_suppression_number), a currency_char (for an all_currency_number), or a sign_char (for an all_sign_number).

16 • I_p /= 0 if p>0.

17 If n < Num'Scale, then the above number is the result of rounding (away from 0 if exactly midway between values).

18 If Blank_When_Zero = True and the displayed magnitude of Item is zero, then the edited output string comprises all space character values. Otherwise, the picture String is treated as a sequence of instances of syntactic categories based on the rules in F.3.1, and the edited output string is the concatenation of string values derived from these categories according to the following mapping rules.

19/5 Table F.1 shows the mapping from a sign control symbol to a corresponding character or string in the edited output. In the columns showing the edited output, a lower-case 'b' represents the space character. If there is no sign control symbol but the value of Item is negative, a layout error occurs and no edited output string is produced.

Table F.1: Edited Output for Sign Control Symbols		
Sign Control Symbol	**Edited Output for Nonnegative Number**	**Edited Output for Negative Number**
'+'	'+'	'–'
'–'	'b'	'–'
'<'	'b'	'('
'>'	'b'	')'
"CR"	"bb"	"CR"
"DB"	"bb"	"DB"

An instance of fixed_LHS_sign maps to a character as shown in Table F.1. 20/5

An instance of fixed_$_char maps to Currency. 21

An instance of direct_insertion maps to Separator if direct_insertion = '_', and to the direct_insertion Character otherwise. 22

An instance of number maps to a string *integer_part* & *radix_part* & *fraction_part* where: 23

- The string for *integer_part* is obtained as follows: 24
  1. Occurrences of '9' in fore_digits of number are replaced from right to left with the decimal digit character values for I_1, ..., I_p, respectively. 25
  2. Each occurrence of '9' in fore_digits to the left of the leftmost '9' replaced according to rule 1 is replaced with '0'. 26
  3. If p exceeds the number of occurrences of '9' in fore_digits of number, then the excess leftmost digits are eligible for use in the mapping of an instance of zero_suppression, floating_LHS_sign, floating_$_currency, or floating_#_currency to the left of number; if there is no such instance, then a layout error occurs and no edited output string is produced. 27
- The *radix_part* is: 28
  - "" if number does not include a radix, if radix = 'V', or if radix = 'v' 29
  - Radix_Mark if number includes '.' as radix 30
- The string for *fraction_part* is obtained as follows: 31
  1. Occurrences of '9' in aft_digits of number are replaced from left to right with the decimal digit character values for F_1, ... F_q. 32
  2. Each occurrence of '9' in aft_digits to the right of the rightmost '9' replaced according to rule 1 is replaced by '0'. 33

An instance of zero_suppression maps to the string obtained as follows: 34

1. The rightmost 'Z', 'z', or '*' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the zero_suppression instance, 35
2. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of some 'Z', 'z', or '*' in zero_suppression that has been mapped to an excess digit, 36

37 3. Each Character to the left of the leftmost Character replaced according to rule 1 above is replaced by:

38 • the space character if the zero suppression Character is 'Z' or 'z', or

39 • the Fill character if the zero suppression Character is '*'.

40 4. A layout error occurs if some excess digits remain after all 'Z', 'z', and '*' Character values in zero_suppression have been replaced via rule 1; no edited output string is produced.

41/5 An instance of RHS_sign maps to a character or string as shown in Table F.1.

42 An instance of floating_LHS_sign maps to the string obtained as follows.

43 1. Up to all but one of the rightmost LHS_Sign Character values are replaced by the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_LHS_sign instance.

44/5 2. The next Character to the left is replaced with the character given by the entry in Table F.1 corresponding to the LHS_Sign Character.

45 3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost LHS_Sign character replaced according to rule 1.

46 4. Any other Character is replaced by the space character.

47 5. A layout error occurs if some excess digits remain after replacement via rule 1; no edited output string is produced.

48 An instance of fixed_#_currency maps to the Currency string with n space character values concatenated on the left (if the instance does not follow a radix) or on the right (if the instance does follow a radix), where n is the difference between the length of the fixed_#_currency instance and Currency'Length. A layout error occurs if Currency'Length exceeds the length of the fixed_#_currency instance; no edited output string is produced.

49 An instance of floating_$_currency maps to the string obtained as follows:

50 1. Up to all but one of the rightmost '$' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_$_currency instance.

51 2. The next Character to the left is replaced by the Currency string.

52 3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '$' Character replaced via rule 1.

53 4. Each other Character is replaced by the space character.

54 5. A layout error occurs if some excess digits remain after replacement by rule 1; no edited output string is produced.

55 An instance of floating_#_currency maps to the string obtained as follows:

56 1. Up to all but one of the rightmost '#' Character values are replaced with the excess digits (if any) from the *integer_part* of the mapping of the number to the right of the floating_#_currency instance.

57 2. The substring whose last Character occurs at the position immediately preceding the leftmost Character replaced via rule 1, and whose length is Currency'Length, is replaced by the Currency string.

3. A context_sensitive_insertion Character is replaced as though it were a direct_insertion Character, if it occurs to the right of the leftmost '#' replaced via rule 1. 58

4. Any other Character is replaced by the space character. 59

5. A layout error occurs if some excess digits remain after replacement rule 1, or if there is no substring with the required length for replacement rule 2; no edited output string is produced. 60

An instance of all_zero_suppression_number maps to: 61

- a string of all spaces if the displayed magnitude of Item is zero, the zero_suppression_char is 'Z' or 'z', and the instance of all_zero_suppression_number does not have a radix at its last character position; 62

- a string containing the Fill character in each position except for the character (if any) corresponding to radix, if zero_suppression_char = '*' and the displayed magnitude of Item is zero; 63

- otherwise, the same result as if each zero_suppression_char in all_zero_suppression_aft were '9', interpreting the instance of all_zero_suppression_number as either zero_suppression number (if a radix and all_zero_suppression_aft are present), or as zero_suppression otherwise. 64

An instance of all_sign_number maps to: 65

- a string of all spaces if the displayed magnitude of Item is zero and the instance of all_sign_number does not have a radix at its last character position; 66

- otherwise, the same result as if each sign_char in all_sign_number_aft were '9', interpreting the instance of all_sign_number as either floating_LHS_sign number (if a radix and all_sign_number_aft are present), or as floating_LHS_sign otherwise. 67

An instance of all_currency_number maps to: 68

- a string of all spaces if the displayed magnitude of Item is zero and the instance of all_currency_number does not have a radix at its last character position; 69

- otherwise, the same result as if each currency_char in all_currency_number_aft were '9', interpreting the instance of all_currency_number as floating_$_currency number or floating_#_currency number (if a radix and all_currency_number_aft are present), or as floating_$_currency or floating_#_currency otherwise. 70

Examples

Examples of use of edited output; in the result string values shown below, 'b' represents the space character: 71/5

```
Item:          Picture and Result Strings:                                    72
123456.78      Picture:  "-###**_***_**9.99"                                  73/3
               Result:   "bbb$***123,456.78"
                         "bbFF***123.456,78" (currency = "FF",
                                              separator = '.',
                                              radix mark = ',')
123456.78      Picture:  "-$**_***_**9.99"                                    74/1
               Result:   "b$***123,456.78"
                        "bFF***123.456,78" (currency = "FF",
                                            separator = '.',
                                            radix mark = ',')
0.0          Picture: "-$$$$$$.$$"                                           75
             Result:  "bbbbbbbbbb"
0.20         Picture: "-$$$$$$.$$"                                           76
             Result:  "bbbbbb$.20"
```

```
-1234.565    Picture: "<<<<_<<<.<<###>"
             Result:  "bb(1,234.57DMb)"  (currency = "DM")

12345.67     Picture: "###_###_##9.99"
             Result:  "bbCHF12,345.67"   (currency = "CHF")
```

F.3.3 The Package Text_IO.Editing

1 The package Text_IO.Editing provides a private type Picture with associated operations, and a generic package Decimal_Output. An object of type Picture is composed from a well-formed picture String (see F.3.1) and a Boolean item indicating whether a zero numeric value will result in an edited output string of all space characters. The package Decimal_Output contains edited output subprograms implementing the effects defined in F.3.2.

Static Semantics

2 The library package Text_IO.Editing has the following declaration:

3/5
```
package Ada.Text_IO.Editing
   with Nonblocking, Global => in out synchronized is

4     type Picture is private;

5     function Valid (Pic_String      : in String;
                      Blank_When_Zero : in Boolean := False) return Boolean;

6     function To_Picture (Pic_String      : in String;
                           Blank_When_Zero : in Boolean := False)
         return Picture;

7     function Pic_String      (Pic : in Picture) return String;
      function Blank_When_Zero (Pic : in Picture) return Boolean;

8     Max_Picture_Length  : constant := implementation_defined;

9     Picture_Error       : exception;

10    Default_Currency    : constant String    := "$";
      Default_Fill        : constant Character := '*';
      Default_Separator   : constant Character := ',';
      Default_Radix_Mark  : constant Character := '.';

11    generic
         type Num is delta <> digits <>;
         Default_Currency   : in String    := Text_IO.Editing.Default_Currency;
         Default_Fill       : in Character := Text_IO.Editing.Default_Fill;
         Default_Separator  : in Character :=
                                 Text_IO.Editing.Default_Separator;
         Default_Radix_Mark : in Character :=
                                 Text_IO.Editing.Default_Radix_Mark;
      package Decimal_Output is
         function Length (Pic      : in Picture;
                          Currency : in String := Default_Currency)
            return Natural;

12       function Valid (Item     : in Num;
                         Pic      : in Picture;
                         Currency : in String := Default_Currency)
            return Boolean;

13       function Image (Item       : in Num;
                         Pic        : in Picture;
                         Currency   : in String    := Default_Currency;
                         Fill       : in Character := Default_Fill;
                         Separator  : in Character := Default_Separator;
                         Radix_Mark : in Character := Default_Radix_Mark)
            return String;
```

```
      procedure Put (File       : in File_Type;                                    14/5
                     Item       : in Num;
                     Pic        : in Picture;
                     Currency   : in String    := Default_Currency;
                     Fill       : in Character := Default_Fill;
                     Separator  : in Character := Default_Separator;
                     Radix_Mark : in Character := Default_Radix_Mark)
         with Nonblocking => False;
      procedure Put (Item       : in Num;                                          15/5
                     Pic        : in Picture;
                     Currency   : in String    := Default_Currency;
                     Fill       : in Character := Default_Fill;
                     Separator  : in Character := Default_Separator;
                     Radix_Mark : in Character := Default_Radix_Mark)
         with Nonblocking => False;
      procedure Put (To         : out String;                                      16
                     Item       : in Num;
                     Pic        : in Picture;
                     Currency   : in String    := Default_Currency;
                     Fill       : in Character := Default_Fill;
                     Separator  : in Character := Default_Separator;
                     Radix_Mark : in Character := Default_Radix_Mark);
   end Decimal_Output;
private
   ... -- not specified by the language
end Ada.Text_IO.Editing;
```

The exception Constraint_Error is raised if the Image function or any of the Put procedures is invoked with a null string for Currency. 17

```
function Valid (Pic_String      : in String;                                       18
                Blank_When_Zero : in Boolean := False) return Boolean;
```

Valid returns True if Pic_String is a well-formed picture String (see F.3.1) the length of whose expansion does not exceed Max_Picture_Length, and if either Blank_When_Zero is False or Pic_String contains no '*'. 19

```
function To_Picture (Pic_String      : in String;                                   20
                     Blank_When_Zero : in Boolean := False)
   return Picture;
```

To_Picture returns a result Picture such that the application of the function Pic_String to this result yields an expanded picture String equivalent to Pic_String, and such that Blank_When_Zero applied to the result Picture is the same value as the parameter Blank_When_Zero. Picture_Error is raised if not Valid(Pic_String, Blank_When_Zero). 21

```
function Pic_String      (Pic : in Picture) return String;                         22

function Blank_When_Zero (Pic : in Picture) return Boolean;
```

If Pic is To_Picture(String_Item, Boolean_Item) for some String_Item and Boolean_Item, then: 23

- Pic_String(Pic) returns an expanded picture String equivalent to String_Item and with any lower-case letter replaced with its corresponding upper-case form, and 24
- Blank_When_Zero(Pic) returns Boolean_Item. 25

If Pic_1 and Pic_2 are objects of type Picture, then "="(Pic_1, Pic_2) is True when 26

- Pic_String(Pic_1) = Pic_String(Pic_2), and 27
- Blank_When_Zero(Pic_1) = Blank_When_Zero(Pic_2). 28

29
```
function Length (Pic      : in Picture;
                 Currency : in String := Default_Currency)
   return Natural;
```

30 Length returns Pic_String(Pic)'Length + Currency_Length_Adjustment – Radix_Adjustment where

31 • Currency_Length_Adjustment =

32 • Currency'Length – 1 if there is some occurrence of '$' in Pic_String(Pic), and

33 • 0 otherwise.

34 • Radix_Adjustment =

35 • 1 if there is an occurrence of 'V' or 'v' in Pic_Str(Pic), and

36 • 0 otherwise.

37
```
function Valid (Item     : in Num;
                Pic      : in Picture;
                Currency : in String := Default_Currency)
   return Boolean;
```

38 Valid returns True if Image(Item, Pic, Currency) does not raise Layout_Error, and returns False otherwise.

39
```
function Image (Item       : in Num;
                Pic        : in Picture;
                Currency   : in String    := Default_Currency;
                Fill       : in Character := Default_Fill;
                Separator  : in Character := Default_Separator;
                Radix_Mark : in Character := Default_Radix_Mark)
   return String;
```

40 Image returns the edited output String as defined in F.3.2 for Item, Pic_String(Pic), Blank_When_Zero(Pic), Currency, Fill, Separator, and Radix_Mark. If these rules identify a layout error, then Image raises the exception Layout_Error.

41
```
procedure Put (File       : in File_Type;
               Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);

procedure Put (Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);
```

42 Each of these Put procedures outputs Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) consistent with the conventions for Put for other real types in case of bounded line length (see A.10.6, “Get and Put Procedures”).

43
```
procedure Put (To         : out String;
               Item       : in Num;
               Pic        : in Picture;
               Currency   : in String    := Default_Currency;
               Fill       : in Character := Default_Fill;
               Separator  : in Character := Default_Separator;
               Radix_Mark : in Character := Default_Radix_Mark);
```

Put copies Image(Item, Pic, Currency, Fill, Separator, Radix_Mark) to the given string, right justified. Otherwise, unassigned Character values in To are assigned the space character. If To'Length is less than the length of the string resulting from Image, then Layout_Error is raised. 44/3

Implementation Requirements

Max_Picture_Length shall be at least 30. The implementation shall support currency strings of length up to at least 10, both for Default_Currency in an instantiation of Decimal_Output, and for Currency in an invocation of Image or any of the Put procedures. 45

F.3.4 The Package Wide_Text_IO.Editing

Static Semantics

The child package Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that: 1

- each occurrence of Character is replaced by Wide_Character, 2
- each occurrence of Text_IO is replaced by Wide_Text_IO, 3
- the subtype of Default_Currency is Wide_String rather than String, and 4
- each occurrence of String in the generic package Decimal_Output is replaced by Wide_String. 5

NOTE Each of the functions Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_String) as its parameter or result subtype, since a picture String is not localizable. 6

F.3.5 The Package Wide_Wide_Text_IO.Editing

Static Semantics

The child package Wide_Wide_Text_IO.Editing has the same contents as Text_IO.Editing, except that: 1/2

- each occurrence of Character is replaced by Wide_Wide_Character, 2/2
- each occurrence of Text_IO is replaced by Wide_Wide_Text_IO, 3/2
- the subtype of Default_Currency is Wide_Wide_String rather than String, and 4/2
- each occurrence of String in the generic package Decimal_Output is replaced by Wide_Wide_String. 5/2

NOTE Each of the functions Wide_Wide_Text_IO.Editing.Valid, To_Picture, and Pic_String has String (versus Wide_Wide_String) as its parameter or result subtype, since a picture String is not localizable. 6/2

Annex G
(normative)
Numerics

The Numerics Annex specifies 1

- features for complex arithmetic, including complex I/O; 2
- a mode (“strict mode”), in which the predefined arithmetic operations of floating point and fixed point types and the functions and operations of various predefined packages have to provide guaranteed accuracy or conform to other numeric performance requirements, which the Numerics Annex also specifies; 3
- a mode (“relaxed mode”), in which there are no accuracy or other numeric performance requirements to be satisfied, as for implementations not conforming to the Numerics Annex; 4/5
- models of floating point and fixed point arithmetic on which the accuracy requirements of strict mode are based; 5/2
- the definitions of the model-oriented attributes of floating point types that apply in the strict mode; and 6/2
- features for the manipulation of real and complex vectors and matrices. 6.1/2

Implementation Advice

If Fortran (respectively, C) is widely supported in the target environment, implementations supporting the Numerics Annex should provide the child package Interfaces.Fortran (respectively, Interfaces.C) specified in Annex B and should support a *convention*_identifier of Fortran (respectively, C) for the Convention aspect (see Annex B), thus allowing Ada programs to interface with programs written in that language. 7/3

G.1 Complex Arithmetic

Types and arithmetic operations for complex arithmetic are provided in Generic_Complex_Types, which is defined in G.1.1. Implementation-defined approximations to the complex analogs of the mathematical functions known as the “elementary functions” are provided by the subprograms in Generic_Complex_-Elementary_Functions, which is defined in G.1.2. Both of these library units are generic children of the predefined package Numerics (see A.5). Nongeneric equivalents of these generic packages for each of the predefined floating point types are also provided as children of Numerics. 1

G.1.1 Complex Types

Static Semantics

The generic library package Numerics.Generic_Complex_Types has the following declaration: 1

```
generic                                                                 2/5
   type Real is digits <>;
package Ada.Numerics.Generic_Complex_Types
   with Pure, Nonblocking is

   type Complex is                                                      3
      record
         Re, Im : Real'Base;
      end record;
```

4/5
```
type Imaginary is private
   with Preelaborable_Initialization;
```

5
```
i : constant Imaginary;
j : constant Imaginary;
```

6
```
function Re (X : Complex)   return Real'Base;
function Im (X : Complex)   return Real'Base;
function Im (X : Imaginary) return Real'Base;
```

7
```
procedure Set_Re (X  : in out Complex;
                  Re : in     Real'Base);
procedure Set_Im (X  : in out Complex;
                  Im : in     Real'Base);
procedure Set_Im (X  :    out Imaginary;
                  Im : in     Real'Base);
```

8
```
function Compose_From_Cartesian (Re, Im : Real'Base) return Complex;
function Compose_From_Cartesian (Re     : Real'Base) return Complex;
function Compose_From_Cartesian (Im     : Imaginary) return Complex;
```

9
```
function Modulus (X     : Complex) return Real'Base;
function "abs"   (Right : Complex) return Real'Base renames Modulus;
```

10
```
function Argument (X     : Complex)   return Real'Base;
function Argument (X     : Complex;
                   Cycle : Real'Base) return Real'Base;
```

11
```
function Compose_From_Polar (Modulus, Argument        : Real'Base)
   return Complex;
function Compose_From_Polar (Modulus, Argument, Cycle : Real'Base)
   return Complex;
```

12
```
function "+"       (Right : Complex) return Complex;
function "-"       (Right : Complex) return Complex;
function Conjugate (X     : Complex) return Complex;
```

13
```
function "+" (Left, Right : Complex) return Complex;
function "-" (Left, Right : Complex) return Complex;
function "*" (Left, Right : Complex) return Complex;
function "/" (Left, Right : Complex) return Complex;
```

14
```
function "**" (Left : Complex; Right : Integer) return Complex;
```

15
```
function "+"       (Right : Imaginary) return Imaginary;
function "-"       (Right : Imaginary) return Imaginary;
function Conjugate (X     : Imaginary) return Imaginary renames "-";
function "abs"     (Right : Imaginary) return Real'Base;
```

16
```
function "+" (Left, Right : Imaginary) return Imaginary;
function "-" (Left, Right : Imaginary) return Imaginary;
function "*" (Left, Right : Imaginary) return Real'Base;
function "/" (Left, Right : Imaginary) return Real'Base;
```

17
```
function "**" (Left : Imaginary; Right : Integer) return Complex;
```

18
```
function "<"  (Left, Right : Imaginary) return Boolean;
function "<=" (Left, Right : Imaginary) return Boolean;
function ">"  (Left, Right : Imaginary) return Boolean;
function ">=" (Left, Right : Imaginary) return Boolean;
```

19
```
function "+" (Left : Complex;   Right : Real'Base) return Complex;
function "+" (Left : Real'Base; Right : Complex)   return Complex;
function "-" (Left : Complex;   Right : Real'Base) return Complex;
function "-" (Left : Real'Base; Right : Complex)   return Complex;
function "*" (Left : Complex;   Right : Real'Base) return Complex;
function "*" (Left : Real'Base; Right : Complex)   return Complex;
function "/" (Left : Complex;   Right : Real'Base) return Complex;
function "/" (Left : Real'Base; Right : Complex)   return Complex;
```

```
   function "+" (Left : Complex;   Right : Imaginary) return Complex;
   function "+" (Left : Imaginary; Right : Complex)   return Complex;
   function "-" (Left : Complex;   Right : Imaginary) return Complex;
   function "-" (Left : Imaginary; Right : Complex)   return Complex;
   function "*" (Left : Complex;   Right : Imaginary) return Complex;
   function "*" (Left : Imaginary; Right : Complex)   return Complex;
   function "/" (Left : Complex;   Right : Imaginary) return Complex;
   function "/" (Left : Imaginary; Right : Complex)   return Complex;
```

20

```
   function "+" (Left : Imaginary; Right : Real'Base) return Complex;
   function "+" (Left : Real'Base; Right : Imaginary) return Complex;
   function "-" (Left : Imaginary; Right : Real'Base) return Complex;
   function "-" (Left : Real'Base; Right : Imaginary) return Complex;
   function "*" (Left : Imaginary; Right : Real'Base) return Imaginary;
   function "*" (Left : Real'Base; Right : Imaginary) return Imaginary;
   function "/" (Left : Imaginary; Right : Real'Base) return Imaginary;
   function "/" (Left : Real'Base; Right : Imaginary) return Imaginary;
```

21

```
private
```

22

```
   type Imaginary is new Real'Base;
   i : constant Imaginary := 1.0;
   j : constant Imaginary := 1.0;
```

23

```
end Ada.Numerics.Generic_Complex_Types;
```

24

The library package Numerics.Complex_Types is declared pure and defines the same types, constants, and subprograms as Numerics.Generic_Complex_Types, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents of Numerics.Generic_Complex_Types for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Types, Numerics.Long_Complex_Types, etc. 25/1

Complex is a visible type with Cartesian components. 26/2

Imaginary is a private type; its full type is derived from Real'Base. 27

The arithmetic operations and the Re, Im, Modulus, Argument, and Conjugate functions have their usual mathematical meanings. When applied to a parameter of pure-imaginary type, the “imaginary-part” function Im yields the value of its parameter, as the corresponding real value. The remaining subprograms have the following meanings: 28

- The Set_Re and Set_Im procedures replace the designated component of a complex parameter with the given real value; applied to a parameter of pure-imaginary type, the Set_Im procedure replaces the value of that parameter with the imaginary value corresponding to the given real value. 29
- The Compose_From_Cartesian function constructs a complex value from the given real and imaginary components. If only one component is given, the other component is implicitly zero. 30
- The Compose_From_Polar function constructs a complex value from the given modulus (radius) and argument (angle). When the value of the parameter Modulus is positive (resp., negative), the result is the complex value represented by the point in the complex plane lying at a distance from the origin given by the absolute value of Modulus and forming an angle measured counterclockwise from the positive (resp., negative) real axis given by the value of the parameter Argument. 31

When the Cycle parameter is specified, the result of the Argument function and the parameter Argument of the Compose_From_Polar function are measured in units such that a full cycle of revolution has the given value; otherwise, they are measured in radians. 32

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply the principal branch: 33

- The result of the Modulus function is nonnegative.
- The result of the Argument function is in the quadrant containing the point in the complex plane represented by the parameter X. This may be any quadrant (I through IV); thus, the range of the Argument function is approximately $-\pi$ to π (–Cycle/2.0 to Cycle/2.0, if the parameter Cycle is specified). When the point represented by the parameter X lies on the negative real axis, the result approximates
  - π (resp., $-\pi$) when the sign of the imaginary component of X is positive (resp., negative), if Real'Signed_Zeros is True;
  - π, if Real'Signed_Zeros is False.
- Because a result lying on or near one of the axes may not be exactly representable, the approximation inherent in computing the result may place it in an adjacent quadrant, close to but on the wrong side of the axis.

Dynamic Semantics

The exception Numerics.Argument_Error is raised by the Argument and Compose_From_Polar functions with specified cycle, signaling a parameter value outside the domain of the corresponding mathematical function, when the value of the parameter Cycle is zero or negative.

The exception Constraint_Error is raised by the division operator when the value of the right operand is zero, and by the exponentiation operator when the value of the left operand is zero and the value of the exponent is negative, provided that Real'Machine_Overflows is True; when Real'Machine_Overflows is False, the result is unspecified. Constraint_Error can also be raised when a finite result overflows (see G.2.6).

Implementation Requirements

In the implementation of Numerics.Generic_Complex_Types, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Real.

In the following cases, evaluation of a complex arithmetic operation shall yield the *prescribed result*, provided that the preceding rules do not call for an exception to be raised:

- The results of the Re, Im, and Compose_From_Cartesian functions are exact.
- The real (resp., imaginary) component of the result of a binary addition operator that yields a result of complex type is exact when either of its operands is of pure-imaginary (resp., real) type.
- The real (resp., imaginary) component of the result of a binary subtraction operator that yields a result of complex type is exact when its right operand is of pure-imaginary (resp., real) type.
- The real component of the result of the Conjugate function for the complex type is exact.
- When the point in the complex plane represented by the parameter X lies on the nonnegative real axis, the Argument function yields a result of zero.
- When the value of the parameter Modulus is zero, the Compose_From_Polar function yields a result of zero.
- When the value of the parameter Argument is equal to a multiple of the quarter cycle, the result of the Compose_From_Polar function with specified cycle lies on one of the axes. In this case, one of its components is zero, and the other has the magnitude of the parameter Modulus.
- Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand. Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero, provided that the exponent is nonzero.

When the left operand is of pure-imaginary type, one component of the result of the exponentiation operator is zero.

When the result, or a result component, of any operator of Numerics.Generic_Complex_Types has a mathematical definition in terms of a single arithmetic or relational operation, that result or result component exhibits the accuracy of the corresponding operation of the type Real. 51

Other accuracy requirements for the Modulus, Argument, and Compose_From_Polar functions, and accuracy requirements for the multiplication of a pair of complex operands or for division by a complex operand, all of which apply only in the strict mode, are given in G.2.6. 52

The sign of a zero result or zero result component yielded by a complex arithmetic operation or function is implementation defined when Real'Signed_Zeros is True. 53

Implementation Permissions

The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required. 54/5

Implementations may obtain the result of exponentiation of a complex or pure-imaginary operand by repeated complex multiplication, with arbitrary association of the factors and with a possible final complex reciprocation (when the exponent is negative). Implementations are also permitted to obtain the result of exponentiation of a complex operand, but not of a pure-imaginary operand, by converting the left operand to a polar representation; exponentiating the modulus by the given exponent; multiplying the argument by the given exponent; and reconverting to a Cartesian representation. Because of this implementation freedom, no accuracy requirement is imposed on complex exponentiation (except for the prescribed results given above, which apply regardless of the implementation method chosen). 55/2

Implementation Advice

Because the usual mathematical meaning of multiplication of a complex operand and a real operand is that of the scaling of both components of the former by the latter, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex multiplication. In systems that, in the future, support an Ada binding to ISO/IEC 60559:2020, the latter technique will not generate the required result when one of the components of the complex operand is infinite. (Explicit multiplication of the infinite component by the zero component obtained during promotion yields a NaN that propagates into the final result.) Analogous advice applies in the case of multiplication of a complex operand and a pure-imaginary operand, and in the case of division of a complex operand by a real or pure-imaginary operand. 56/5

Likewise, because the usual mathematical meaning of addition of a complex operand and a real operand is that the imaginary operand remains unchanged, an implementation should not perform this operation by first promoting the real operand to complex type and then performing a full complex addition. In implementations in which the Signed_Zeros attribute of the component type is True (and which therefore conform to ISO/IEC 60559:2020 in regard to the handling of the sign of zero in predefined arithmetic operations), the latter technique will not generate the required result when the imaginary component of the complex operand is a negatively signed zero. (Explicit addition of the negative zero to the zero obtained during promotion yields a positive zero.) Analogous advice applies in the case of addition of a complex operand and a pure-imaginary operand, and in the case of subtraction of a complex operand and a real or pure-imaginary operand. 57/5

Implementations in which Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. As one example, the result of the Argument function should have the sign of the imaginary component of the parameter X when the point represented by that 58

parameter lies on the positive real axis; as another, the sign of the imaginary component of the Compose_-From_Polar function should be the same as (resp., the opposite of) that of the Argument parameter when that parameter has a value of zero and the Modulus parameter has a nonnegative (resp., negative) value.

G.1.2 Complex Elementary Functions

Static Semantics

1 The generic library package Numerics.Generic_Complex_Elementary_Functions has the following declaration:

2/5
```
with Ada.Numerics.Generic_Complex_Types;
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Elementary_Functions
   with Pure, Nonblocking is
```

3
```
   function Sqrt (X : Complex)   return Complex;
   function Log  (X : Complex)   return Complex;
   function Exp  (X : Complex)   return Complex;
   function Exp  (X : Imaginary) return Complex;
   function "**" (Left : Complex;   Right : Complex)   return Complex;
   function "**" (Left : Complex;   Right : Real'Base) return Complex;
   function "**" (Left : Real'Base; Right : Complex)   return Complex;
```

4
```
   function Sin (X : Complex) return Complex;
   function Cos (X : Complex) return Complex;
   function Tan (X : Complex) return Complex;
   function Cot (X : Complex) return Complex;
```

5
```
   function Arcsin (X : Complex) return Complex;
   function Arccos (X : Complex) return Complex;
   function Arctan (X : Complex) return Complex;
   function Arccot (X : Complex) return Complex;
```

6
```
   function Sinh (X : Complex) return Complex;
   function Cosh (X : Complex) return Complex;
   function Tanh (X : Complex) return Complex;
   function Coth (X : Complex) return Complex;
```

7
```
   function Arcsinh (X : Complex) return Complex;
   function Arccosh (X : Complex) return Complex;
   function Arctanh (X : Complex) return Complex;
   function Arccoth (X : Complex) return Complex;
```

8
```
end Ada.Numerics.Generic_Complex_Elementary_Functions;
```

9/1 The library package Numerics.Complex_Elementary_Functions is declared pure and defines the same subprograms as Numerics.Generic_Complex_Elementary_Functions, except that the predefined type Float is systematically substituted for Real'Base, and the Complex and Imaginary types exported by Numerics.-Complex_Types are systematically substituted for Complex and Imaginary, throughout. Nongeneric equivalents of Numerics.Generic_Complex_Elementary_Functions corresponding to each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_-Elementary_Functions, Numerics.Long_Complex_Elementary_Functions, etc.

10 The overloading of the Exp function for the pure-imaginary type is provided to give the user an alternate way to compose a complex value from a given modulus and argument. In addition to Compose_From_-Polar(Rho, Theta) (see G.1.1), the programmer may write Rho * Exp(i * Theta).

11 The imaginary (resp., real) component of the parameter X of the forward hyperbolic (resp., trigonometric) functions and of the Exp function (and the parameter X, itself, in the case of the overloading of the Exp

function for the pure-imaginary type) represents an angle measured in radians, as does the imaginary (resp., real) component of the result of the Log and inverse hyperbolic (resp., trigonometric) functions.

The functions have their usual mathematical meanings. However, the arbitrariness inherent in the placement of branch cuts, across which some of the complex elementary functions exhibit discontinuities, is eliminated by the following conventions: 12

- The imaginary component of the result of the Sqrt and Log functions is discontinuous as the parameter X crosses the negative real axis. 13
- The result of the exponentiation operator when the left operand is of complex type is discontinuous as that operand crosses the negative real axis. 14
- The imaginary component of the result of the Arcsin, Arccos, and Arctanh functions is discontinuous as the parameter X crosses the real axis to the left of –1.0 or the right of 1.0. 15/2
- The real component of the result of the Arctan and Arcsinh functions is discontinuous as the parameter X crosses the imaginary axis below $-i$ or above i. 16/2
- The real component of the result of the Arccot function is discontinuous as the parameter X crosses the imaginary axis below $-i$ or above i. 17/2
- The imaginary component of the Arccosh function is discontinuous as the parameter X crosses the real axis to the left of 1.0. 18
- The imaginary component of the result of the Arccoth function is discontinuous as the parameter X crosses the real axis between –1.0 and 1.0. 19

The computed results of the mathematically multivalued functions are rendered single-valued by the following conventions, which are meant to imply that the principal branch is an analytic continuation of the corresponding real-valued function in Numerics.Generic_Elementary_Functions. (For Arctan and Arccot, the single-argument function in question is that obtained from the two-argument version by fixing the second argument to be its default value.) 20/2

- The real component of the result of the Sqrt and Arccosh functions is nonnegative. 21
- The same convention applies to the imaginary component of the result of the Log function as applies to the result of the natural-cycle version of the Argument function of Numerics.Generic_Complex_Types (see G.1.1). 22
- The range of the real (resp., imaginary) component of the result of the Arcsin and Arctan (resp., Arcsinh and Arctanh) functions is approximately $-\pi/2.0$ to $\pi/2.0$. 23
- The real (resp., imaginary) component of the result of the Arccos and Arccot (resp., Arccoth) functions ranges from 0.0 to approximately π. 24
- The range of the imaginary component of the result of the Arccosh function is approximately $-\pi$ to π. 25

In addition, the exponentiation operator inherits the single-valuedness of the Log function. 26

Dynamic Semantics

The exception Numerics.Argument_Error is raised by the exponentiation operator, signaling a parameter value outside the domain of the corresponding mathematical function, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is zero. 27

The exception Constraint_Error is raised, signaling a pole of the mathematical function (analogous to dividing by zero), in the following cases, provided that Complex_Types.Real'Machine_Overflows is True: 28

- by the Log, Cot, and Coth functions, when the value of the parameter X is zero; 29

30 • by the exponentiation operator, when the value of the left operand is zero and the real component of the exponent (or the exponent itself, when it is of real type) is negative;

31 • by the Arctan and Arccot functions, when the value of the parameter X is ± *i*;

32 • by the Arctanh and Arccoth functions, when the value of the parameter X is ± 1.0.

33 Constraint_Error can also be raised when a finite result overflows (see G.2.6); this may occur for parameter values sufficiently *near* poles, and, in the case of some of the functions, for parameter values having components of sufficiently large magnitude. When Complex_Types.Real'Machine_Overflows is False, the result at poles is unspecified.

Implementation Requirements

34 In the implementation of Numerics.Generic_Complex_Elementary_Functions, the range of intermediate values allowed during the calculation of a final result shall not be affected by any range constraint of the subtype Complex_Types.Real.

35 In the following cases, evaluation of a complex elementary function shall yield the *prescribed result* (or a result having the prescribed component), provided that the preceding rules do not call for an exception to be raised:

36 • When the parameter X has the value zero, the Sqrt, Sin, Arcsin, Tan, Arctan, Sinh, Arcsinh, Tanh, and Arctanh functions yield a result of zero; the Exp, Cos, and Cosh functions yield a result of one; the Arccos and Arccot functions yield a real result; and the Arccoth function yields an imaginary result.

37 • When the parameter X has the value one, the Sqrt function yields a result of one; the Log, Arccos, and Arccosh functions yield a result of zero; and the Arcsin function yields a real result.

38 • When the parameter X has the value –1.0, the Sqrt function yields the result

39 – *i* (resp., –*i*), when the sign of the imaginary component of X is positive (resp., negative), if Complex_Types.Real'Signed_Zeros is True;

40 – *i*, if Complex_Types.Real'Signed_Zeros is False;

41/2 • When the parameter X has the value –1.0, the Log function yields an imaginary result; and the Arcsin and Arccos functions yield a real result.

42 • When the parameter X has the value ± *i*, the Log function yields an imaginary result.

43 • Exponentiation by a zero exponent yields the value one. Exponentiation by a unit exponent yields the value of the left operand (as a complex value). Exponentiation of the value one yields the value one. Exponentiation of the value zero yields the value zero.

44 Other accuracy requirements for the complex elementary functions, which apply only in the strict mode, are given in G.2.6.

45 The sign of a zero result or zero result component yielded by a complex elementary function is implementation defined when Complex_Types.Real'Signed_Zeros is True.

Implementation Permissions

46/5 The nongeneric equivalent packages can be actual instantiations of the generic package with the appropriate predefined nongeneric equivalent of Numerics.Generic_Complex_Types, though that is not required; if they are, then the latter shall have been obtained by actual instantiation of Numerics.Generic_Complex_Types.

The exponentiation operator may be implemented in terms of the Exp and Log functions. Because this implementation yields poor accuracy in some parts of the domain, no accuracy requirement is imposed on complex exponentiation. 47

The implementation of the Exp function of a complex parameter X is allowed to raise the exception Constraint_Error, signaling overflow, when the real component of X exceeds an unspecified threshold that is approximately log(Complex_Types.Real'Safe_Last). This permission recognizes the impracticality of avoiding overflow in the marginal case that the exponential of the real component of X exceeds the safe range of Complex_Types.Real but both components of the final result do not. Similarly, the Sin and Cos (resp., Sinh and Cosh) functions are allowed to raise the exception Constraint_Error, signaling overflow, when the absolute value of the imaginary (resp., real) component of the parameter X exceeds an unspecified threshold that is approximately log(Complex_Types.Real'Safe_Last) + log(2.0). This permission recognizes the impracticality of avoiding overflow in the marginal case that the hyperbolic sine or cosine of the imaginary (resp., real) component of X exceeds the safe range of Complex_Types.Real but both components of the final result do not. 48

Implementation Advice

Implementations in which Complex_Types.Real'Signed_Zeros is True should attempt to provide a rational treatment of the signs of zero results and result components. For example, many of the complex elementary functions have components that are odd functions of one of the parameter components; in these cases, the result component should have the sign of the parameter component at the origin. Other complex elementary functions have zero components whose sign is opposite that of a parameter component at the origin, or is always positive or always negative. 49

G.1.3 Complex Input-Output

The generic package Text_IO.Complex_IO defines procedures for the formatted input and output of complex values. The generic actual parameter in an instantiation of Text_IO.Complex_IO is an instance of Numerics.Generic_Complex_Types for some floating point subtype. Exceptional conditions are reported by raising the appropriate exception defined in Text_IO. 1

Static Semantics

The generic library package Text_IO.Complex_IO has the following declaration: 2

```
with Ada.Numerics.Generic_Complex_Types;                                3/5
generic
   with package Complex_Types is
         new Ada.Numerics.Generic_Complex_Types (<>);
package Ada.Text_IO.Complex_IO
   with Global => in out synchronized is

   use Complex_Types;                                                   4

   Default_Fore : Field := 2;                                           5
   Default_Aft  : Field := Real'Digits - 1;
   Default_Exp  : Field := 3;

   procedure Get (File  : in  File_Type;                                6
                  Item  : out Complex;
                  Width : in  Field := 0);
   procedure Get (Item  : out Complex;
                  Width : in  Field := 0);
```

```
7          procedure Put (File : in File_Type;
                          Item : in Complex;
                          Fore : in Field := Default_Fore;
                          Aft  : in Field := Default_Aft;
                          Exp  : in Field := Default_Exp);
           procedure Put (Item : in Complex;
                          Fore : in Field := Default_Fore;
                          Aft  : in Field := Default_Aft;
                          Exp  : in Field := Default_Exp);
8/5        procedure Get (From : in  String;
                          Item : out Complex;
                          Last : out Positive)
              with Nonblocking;
           procedure Put (To   : out String;
                          Item : in  Complex;
                          Aft  : in  Field := Default_Aft;
                          Exp  : in  Field := Default_Exp)
              with Nonblocking;
9       end Ada.Text_IO.Complex_IO;
```

9.1/2 The library package Complex_Text_IO defines the same subprograms as Text_IO.Complex_IO, except that the predefined type Float is systematically substituted for Real, and the type Numerics.Complex_Types.Complex is systematically substituted for Complex throughout. Nongeneric equivalents of Text_IO.Complex_IO corresponding to each of the other predefined floating point types are defined similarly, with the names Short_Complex_Text_IO, Long_Complex_Text_IO, etc.

10 The semantics of the Get and Put procedures are as follows:

```
11      procedure Get (File  : in  File_Type;
                       Item  : out Complex;
                       Width : in  Field := 0);
        procedure Get (Item  : out Complex;
                       Width : in  Field := 0);
```

12/1 The input sequence is a pair of optionally signed real literals representing the real and imaginary components of a complex value. These components have the format defined for the corresponding Get procedure of an instance of Text_IO.Float_IO (see A.10.9) for the base subtype of Complex_Types.Real. The pair of components may be separated by a comma or surrounded by a pair of parentheses or both. Blanks are freely allowed before each of the components and before the parentheses and comma, if either is used. If the value of the parameter Width is zero, then

13 • line and page terminators are also allowed in these places;

14 • the components shall be separated by at least one blank or line terminator if the comma is omitted; and

15 • reading stops when the right parenthesis has been read, if the input sequence includes a left parenthesis, or when the imaginary component has been read, otherwise.

15.1 If a nonzero value of Width is supplied, then

16 • the components shall be separated by at least one blank if the comma is omitted; and

17 • exactly Width characters are read, or the characters (possibly none) up to a line terminator, whichever comes first (blanks are included in the count).

18 Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence.

The exception Text_IO.Data_Error is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of Complex_Types.Real. 19

```
procedure Put (File : in File_Type;
               Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
procedure Put (Item : in Complex;
               Fore : in Field := Default_Fore;
               Aft  : in Field := Default_Aft;
               Exp  : in Field := Default_Exp);
```
20

Outputs the value of the parameter Item as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically, 21

- outputs a left parenthesis; 22
- outputs the value of the real component of the parameter Item with the format defined by the corresponding Put procedure of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using the given values of Fore, Aft, and Exp; 23
- outputs a comma; 24
- outputs the value of the imaginary component of the parameter Item with the format defined by the corresponding Put procedure of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using the given values of Fore, Aft, and Exp; 25
- outputs a right parenthesis. 26

```
procedure Get (From : in  String;
               Item : out Complex;
               Last : out Positive);
```
27

Reads a complex value from the beginning of the given string, following the same rule as the Get procedure that reads a complex value from a file, but treating the end of the string as a file terminator. Returns, in the parameter Item, the value of type Complex that corresponds to the input sequence. Returns in Last the index value such that From(Last) is the last character read. 28/2

The exception Text_IO.Data_Error is raised if the input sequence does not have the required syntax or if the components of the complex value obtained are not of the base subtype of Complex_Types.Real. 29

```
procedure Put (To   : out String;
               Item : in  Complex;
               Aft  : in  Field := Default_Aft;
               Exp  : in  Field := Default_Exp);
```
30

Outputs the value of the parameter Item to the given string as a pair of decimal literals representing the real and imaginary components of the complex value, using the syntax of an aggregate. More specifically, 31

- a left parenthesis, the real component, and a comma are left justified in the given string, with the real component having the format defined by the Put procedure (for output to a file) of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using a value of zero for Fore and the given values of Aft and Exp; 32

33 - the imaginary component and a right parenthesis are right justified in the given string, with the imaginary component having the format defined by the Put procedure (for output to a file) of an instance of Text_IO.Float_IO for the base subtype of Complex_Types.Real, using a value for Fore that completely fills the remainder of the string, together with the given values of Aft and Exp.

34 The exception Text_IO.Layout_Error is raised if the given string is too short to hold the formatted output.

Implementation Permissions

35 Other exceptions declared (by renaming) in Text_IO may be raised by the preceding procedures in the appropriate circumstances, as for the corresponding procedures of Text_IO.Float_IO.

G.1.4 The Package Wide_Text_IO.Complex_IO

Static Semantics

1 Implementations shall also provide the generic library package Wide_Text_IO.Complex_IO. Its declaration is obtained from that of Text_IO.Complex_IO by systematically replacing Text_IO by Wide_Text_IO and String by Wide_String; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide characters.

G.1.5 The Package Wide_Wide_Text_IO.Complex_IO

Static Semantics

1/2 Implementations shall also provide the generic library package Wide_Wide_Text_IO.Complex_IO. Its declaration is obtained from that of Text_IO.Complex_IO by systematically replacing Text_IO by Wide_Wide_Text_IO and String by Wide_Wide_String; the description of its behavior is obtained by additionally replacing references to particular characters (commas, parentheses, etc.) by those for the corresponding wide wide characters.

G.2 Numeric Performance Requirements

Implementation Requirements

1 Implementations shall provide a user-selectable mode in which the accuracy and other numeric performance requirements detailed in the following subclauses are observed. This mode, referred to as the *strict mode*, may or may not be the default mode; it directly affects the results of the predefined arithmetic operations of real types and the results of the subprograms in children of the Numerics package, and indirectly affects the operations in other language defined packages. Implementations shall also provide the opposing mode, which is known as the *relaxed mode*.

Implementation Permissions

2 Either mode may be the default mode.

3/5 The two modes can be one and the same.

G.2.1 Model of Floating Point Arithmetic

In the strict mode, the predefined operations of a floating point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described. This behavior is presented in terms of a model of floating point arithmetic that builds on the concept of the canonical form (see A.5.3). 1

Static Semantics

Associated with each floating point type is an infinite set of model numbers. The model numbers of a type are used to define the accuracy requirements that have to be satisfied by certain predefined operations of the type; through certain attributes of the model numbers, they are also used to explain the meaning of a user-declared floating point type declaration. The model numbers of a derived type are those of the parent type; the model numbers of a subtype are those of its type. 2

The *model numbers* of a floating point type T are zero and all the values expressible in the canonical form (for the type T), in which *mantissa* has T'Model_Mantissa digits and *exponent* has a value greater than or equal to T'Model_Emin. (These attributes are defined in G.2.2.) 3

A *model interval* of a floating point type is any interval whose bounds are model numbers of the type. The *model interval* of a type T *associated with a value v* is the smallest model interval of T that includes *v*. (The model interval associated with a model number of a type consists of that number only.) 4

Implementation Requirements

The accuracy requirements for the evaluation of certain predefined operations of floating point types are as follows. 5

An *operand interval* is the model interval, of the type specified for the operand of an operation, associated with the value of the operand. 6

For any predefined arithmetic operation that yields a result of a floating point type T, the required bounds on the result are given by a model interval of T (called the *result interval*) defined in terms of the operand values as follows: 7

- The result interval is the smallest model interval of T that includes the minimum and the maximum of all the values obtained by applying the (exact) mathematical operation to values arbitrarily selected from the respective operand intervals. 8

The result interval of an exponentiation is obtained by applying the above rule to the sequence of multiplications defined by the exponent, assuming arbitrary association of the factors, and to the final division in the case of a negative exponent. 9

The result interval of a conversion of a numeric value to a floating point type T is the model interval of T associated with the operand value, except when the source expression is of a fixed point type with a *small* that is not a power of T'Machine_Radix or is a fixed point multiplication or division either of whose operands has a *small* that is not a power of T'Machine_Radix; in these cases, the result interval is implementation defined. 10

For any of the foregoing operations, the implementation shall deliver a value that belongs to the result interval when both bounds of the result interval are in the safe range of the result type T, as determined by the values of T'Safe_First and T'Safe_Last; otherwise, 11

- if T'Machine_Overflows is True, the implementation shall either deliver a value that belongs to the result interval or raise Constraint_Error; 12

13 • if T'Machine_Overflows is False, the result is implementation defined.

14/5 For any predefined relation on operands of a floating point type T, the implementation may deliver any value (that is, either True or False) obtained by applying the (exact) mathematical comparison to values arbitrarily chosen from the respective operand intervals.

15 The result of a membership test is defined in terms of comparisons of the operand value with the lower and upper bounds of the given range or type mark (the usual rules apply to these comparisons).

Implementation Permissions

16 If the underlying floating point hardware implements division as multiplication by a reciprocal, the result interval for division (and exponentiation by a negative exponent) is implementation defined.

G.2.2 Model-Oriented Attributes of Floating Point Types

1 In implementations that support the Numerics Annex, the model-oriented attributes of floating point types shall yield the values defined here, in both the strict and the relaxed modes. These definitions add conditions to those in A.5.3.

Static Semantics

2 For every subtype S of a floating point type *T*:

3/2 S'Model_Mantissa

Yields the number of digits in the mantissa of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to

3.1/2 $$\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + g$$

3.2/2 where *d* is the requested decimal precision of *T*, and *g* is 0 if *T*'Machine_Radix is a positive power of 10 and 1 otherwise. In addition, *T*'Model_Mantissa shall be less than or equal to the value of *T*'Machine_Mantissa. This attribute yields a value of the type *universal_integer*.

4 S'Model_Emin

Yields the minimum exponent of the canonical form of the model numbers of *T* (see A.5.3). The value of this attribute shall be greater than or equal to the value of *T*'Machine_Emin. This attribute yields a value of the type *universal_integer*.

5 S'Safe_First

Yields the lower bound of the safe range of *T*. The value of this attribute shall be a model number of *T* and greater than or equal to the lower bound of the base range of *T*. In addition, if *T* is declared by a floating_point_definition or is derived from such a type, and the floating_point_definition includes a real_range_specification specifying a lower bound of *lb*, then the value of this attribute shall be less than or equal to *lb*; otherwise, it shall be less than or equal to $-10.0^{4 \cdot d}$, where *d* is the requested decimal precision of *T*. This attribute yields a value of the type *universal_real*.

6 S'Safe_Last

Yields the upper bound of the safe range of *T*. The value of this attribute shall be a model number of *T* and less than or equal to the upper bound of the base range of *T*. In addition, if *T* is declared by a floating_point_definition or is derived from such a type, and the floating_point_definition includes a real_range_specification specifying an upper bound of *ub*, then the value of this attribute shall be greater than or equal to *ub*; otherwise, it shall be greater than or equal to $10.0^{4 \cdot d}$, where d is the requested decimal precision of *T*. This attribute yields a value of the type *universal_real*.

S'Model Denotes a function (of a parameter *X*) whose specification is given in A.5.3. If *X* is a model number of *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent model numbers of *T*. Constraint_Error is raised if the resulting model number is outside the safe range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. 7

Subject to the constraints given above, the values of S'Model_Mantissa and S'Safe_Last are to be maximized, and the values of S'Model_Emin and S'Safe_First minimized, by the implementation as follows: 8

- First, S'Model_Mantissa is set to the largest value for which values of S'Model_Emin, S'Safe_First, and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes. 9
- Next, S'Model_Emin is set to the smallest value for which values of S'Safe_First and S'Safe_Last can be chosen so that the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes and the previously determined value of S'Model_Mantissa. 10
- Finally, S'Safe_First and S'Safe_Last are set (in either order) to the smallest and largest values, respectively, for which the implementation satisfies the strict-mode requirements of G.2.1 in terms of the model numbers and safe range induced by these attributes and the previously determined values of S'Model_Mantissa and S'Model_Emin. 11/3

G.2.3 Model of Fixed Point Arithmetic

In the strict mode, the predefined arithmetic operations of a fixed point type shall satisfy the accuracy requirements specified here and shall avoid or signal overflow in the situations described. 1

Implementation Requirements

The accuracy requirements for the predefined fixed point arithmetic operations and conversions, and the results of relations on fixed point operands, are given below. 2

The operands of the fixed point adding operators, absolute value, and comparisons have the same type. These operations are required to yield exact results, unless they overflow. 3

Multiplications and divisions are allowed between operands of any two fixed point types; the result has to be (implicitly or explicitly) converted to some other numeric type. For purposes of defining the accuracy rules, the multiplication or division and the conversion are treated as a single operation whose accuracy depends on three types (those of the operands and the result). For decimal fixed point types, the attribute T'Round may be used to imply explicit conversion with rounding (see 3.5.10). 4

When the result type is a floating point type, the accuracy is as given in G.2.1. For some combinations of the operand and result types in the remaining cases, the result is required to belong to a small set of values called the *perfect result set*; for other combinations, it is required merely to belong to a generally larger and implementation-defined set of values called the *close result set*. When the result type is a decimal fixed point type, the perfect result set contains a single value; thus, operations on decimal types are always fully specified. 5

When one operand of a fixed-fixed multiplication or division is of type *universal_real*, that operand is not implicitly converted in the usual sense, since the context does not determine a unique target type, but the accuracy of the result of the multiplication or division (that is, whether the result has to belong to the perfect result set or merely the close result set) depends on the value of the operand of type *universal_real* and on the types of the other operand and of the result. 6/5

7 For a fixed point multiplication or division whose (exact) mathematical result is *v*, and for the conversion of a value *v* to a fixed point type, the perfect result set and close result set are defined as follows:

8 • If the result type is an ordinary fixed point type with a *small* of *s*,

9 • if *v* is an integer multiple of *s*, then the perfect result set contains only the value *v*;

10 • otherwise, it contains the integer multiple of *s* just below *v* and the integer multiple of *s* just above *v*.

11 The close result set is an implementation-defined set of consecutive integer multiples of *s* containing the perfect result set as a subset.

12 • If the result type is a decimal type with a *small* of *s*,

13 • if *v* is an integer multiple of *s*, then the perfect result set contains only the value *v*;

14/3 • otherwise, if truncation applies, then it contains only the integer multiple of *s* in the direction toward zero, whereas if rounding applies, then it contains only the nearest integer multiple of *s* (with ties broken by rounding away from zero).

15 The close result set is an implementation-defined set of consecutive integer multiples of *s* containing the perfect result set as a subset.

16 • If the result type is an integer type,

17 • if *v* is an integer, then the perfect result set contains only the value *v*;

18 • otherwise, it contains the integer nearest to the value *v* (if *v* lies equally distant from two consecutive integers, the perfect result set contains the one that is further from zero).

19 The close result set is an implementation-defined set of consecutive integers containing the perfect result set as a subset.

20/5 The result of a fixed point multiplication or division shall belong either to the perfect result set or to the close result set, as described below, if overflow does not occur. In the following cases, if the result type is a fixed point type, let *s* be its *small*; otherwise, that is when the result type is an integer type, let *s* be 1.0.

21/5 • For a multiplication or division neither of whose operands is of type *universal_real*, let *l* and *r* be the *smalls* of the left and right operands. For a multiplication, if $(l \cdot r) / s$ is an integer or the reciprocal of an integer (the *smalls* are said to be "compatible" in this case), the result shall belong to the perfect result set; otherwise, it belongs to the close result set. For a division, if $l / (r \cdot s)$ is an integer or the reciprocal of an integer (that is, the *smalls* are compatible), the result shall belong to the perfect result set; otherwise, it belongs to the close result set.

22 • For a multiplication or division having one *universal_real* operand with a value of *v*, note that it is always possible to factor *v* as an integer multiple of a "compatible" *small*, but the integer multiple may be "too big". If there exists a factorization in which that multiple is less than some implementation-defined limit, the result shall belong to the perfect result set; otherwise, it belongs to the close result set.

23/5 A multiplication P * Q of an operand of a fixed point type F by an operand of type Integer, or vice versa, and a division P / Q of an operand of a fixed point type F by an operand of type Integer, are also allowed. In these cases, the result has the type of F; explicit conversion of the result is never required. The accuracy required in these cases is the same as that required for a multiplication F(P * Q) or a division F(P / Q) obtained by interpreting the operand of the integer type to have a fixed point type with a *small* of 1.0.

24 The accuracy of the result of a conversion from an integer or fixed point type to a fixed point type, or from a fixed point type to an integer type, is the same as that of a fixed point multiplication of the source value by a fixed point operand having a *small* of 1.0 and a value of 1.0, as given by the foregoing rules. The result of a conversion from a floating point type to a fixed point type shall belong to the close result set.

The result of a conversion of a *universal_real* operand to a fixed point type shall belong to the perfect result set.

The possibility of overflow in the result of a predefined arithmetic operation or conversion yielding a result of a fixed point type T is analogous to that for floating point types, except for being related to the base range instead of the safe range. If all of the permitted results belong to the base range of T, then the implementation shall deliver one of the permitted results; otherwise, 25

- if T'Machine_Overflows is True, the implementation shall either deliver one of the permitted results or raise Constraint_Error; 26
- if T'Machine_Overflows is False, the result is implementation defined. 27

G.2.4 Accuracy Requirements for the Elementary Functions

In the strict mode, the performance of Numerics.Generic_Elementary_Functions shall be as specified here. 1

Implementation Requirements

When an exception is not raised, the result of evaluating a function in an instance *EF* of Numerics.Generic_Elementary_Functions belongs to a *result interval*, defined as the smallest model interval of *EF*.Float_Type that contains all the values of the form $f \cdot (1.0 + d)$, where f is the exact value of the corresponding mathematical function at the given parameter values, d is a real number, and $|d|$ is less than or equal to the function's *maximum relative error*. The function delivers a value that belongs to the result interval when both of its bounds belong to the safe range of *EF*.Float_Type; otherwise, 2

- if *EF*.Float_Type'Machine_Overflows is True, the function either delivers a value that belongs to the result interval or raises Constraint_Error, signaling overflow; 3
- if *EF*.Float_Type'Machine_Overflows is False, the result is implementation defined. 4

The maximum relative error exhibited by each function is as follows: 5

- $2.0 \cdot$ *EF*.Float_Type'Model_Epsilon, in the case of the Sqrt, Sin, and Cos functions; 6
- $4.0 \cdot$ *EF*.Float_Type'Model_Epsilon, in the case of the Log, Exp, Tan, Cot, and inverse trigonometric functions; and 7
- $8.0 \cdot$ *EF*.Float_Type'Model_Epsilon, in the case of the forward and inverse hyperbolic functions. 8

The maximum relative error exhibited by the exponentiation operator, which depends on the values of the operands, is $(4.0 + |\text{Right} \cdot \log(\text{Left})| / 32.0) \cdot$ *EF*.Float_Type'Model_Epsilon. 9

The maximum relative error given above applies throughout the domain of the forward trigonometric functions when the Cycle parameter is specified. When the Cycle parameter is omitted, the maximum relative error given above applies only when the absolute value of the angle parameter X is less than or equal to some implementation-defined *angle threshold*, which shall be at least *EF*.Float_Type'Machine_-Radix $^{\lfloor EF.\text{Float\_Type'Machine\_Mantissa}/2 \rfloor}$. Beyond the angle threshold, the accuracy of the forward trigonometric functions is implementation defined. 10

The prescribed results specified in A.5.1 for certain functions at particular parameter values take precedence over the maximum relative error bounds; effectively, they narrow to a single value the result interval allowed by the maximum relative error bounds. Additional rules with a similar effect are given by Table G.1 for the inverse trigonometric functions, at particular parameter values for which the mathematical result is possibly not a model number of *EF*.Float_Type (or is, indeed, even transcendental). In each table entry, the values of the parameters are such that the result lies on the axis between two 11/5

quadrants; the corresponding accuracy rule, which takes precedence over the maximum relative error bounds, is that the result interval is the model interval of *EF*.Float_Type associated with the exact mathematical result given in the table.

12/1 *This paragraph was deleted.*

13 The last line of the table is meant to apply when *EF*.Float_Type'Signed_Zeros is False; the two lines just above it, when *EF*.Float_Type'Signed_Zeros is True and the parameter Y has a zero value with the indicated sign.

Table G.1: Tightly Approximated Elementary Function Results

Function	Value of X	Value of Y	Exact Result when Cycle Specified	Exact Result when Cycle Omitted
Arcsin	1.0	n.a.	Cycle/4.0	π/2.0
Arcsin	–1.0	n.a.	–Cycle/4.0	–π/2.0
Arccos	0.0	n.a.	Cycle/4.0	π/2.0
Arccos	–1.0	n.a.	Cycle/2.0	π
Arctan and Arccot	0.0	positive	Cycle/4.0	π/2.0
Arctan and Arccot	0.0	negative	–Cycle/4.0	–π/2.0
Arctan and Arccot	negative	+0.0	Cycle/2.0	π
Arctan and Arccot	negative	–0.0	–Cycle/2.0	–π
Arctan and Arccot	negative	0.0	Cycle/2.0	π

14 The amount by which the result of an inverse trigonometric function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in A.5.1, is limited. The rule is that the result belongs to the smallest model interval of *EF*.Float_Type that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum relative error bounds, effectively narrowing the result interval allowed by them.

15 Finally, the following specifications also take precedence over the maximum relative error bounds:

16 • The absolute value of the result of the Sin, Cos, and Tanh functions never exceeds one.

17 • The absolute value of the result of the Coth function is never less than one.

18 • The result of the Cosh function is never less than one.

Implementation Advice

19 The versions of the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain. For the same reason, the version of Log without a Base parameter should not be implemented by calling the corresponding version with a Base parameter of Numerics.e.

G.2.5 Performance Requirements for Random Number Generation

In the strict mode, the performance of Numerics.Float_Random and Numerics.Discrete_Random shall be as specified here. 1

Implementation Requirements

Two different calls to the time-dependent Reset procedure shall reset the generator to different states, provided that the calls are separated in time by at least one second and not more than fifty years. 2

The implementation's representations of generator states and its algorithms for generating random numbers shall yield a period of at least $2^{31}-2$; much longer periods are desirable but not required. 3

The implementations of Numerics.Float_Random.Random and Numerics.Discrete_Random.Random shall pass at least 85% of the individual trials in a suite of statistical tests. For Numerics.Float_Random, the tests are applied directly to the floating point values generated (that is, they are not converted to integers first), while for Numerics.Discrete_Random they are applied to the generated values of various discrete types. Each test suite performs 6 different tests, with each test repeated 10 times, yielding a total of 60 individual trials. An individual trial is deemed to pass if the chi-square value (or other statistic) calculated for the observed counts or distribution falls within the range of values corresponding to the 2.5 and 97.5 percentage points for the relevant degrees of freedom (that is, it shall be neither too high nor too low). For the purpose of determining the degrees of freedom, measurement categories are combined whenever the expected counts are fewer than 5. 4/5

G.2.6 Accuracy Requirements for Complex Arithmetic

In the strict mode, the performance of Numerics.Generic_Complex_Types and Numerics.Generic_-Complex_Elementary_Functions shall be as specified here. 1

Implementation Requirements

When an exception is not raised, the result of evaluating a real function of an instance *CT* of Numerics.Generic_Complex_Types (that is, a function that yields a value of subtype *CT*.Real'Base or *CT*.Imaginary) belongs to a result interval defined as for a real elementary function (see G.2.4). 2/5

When an exception is not raised, each component of the result of evaluating a complex function of such an instance, or of an instance of Numerics.Generic_Complex_Elementary_Functions obtained by instantiating the latter with *CT* (that is, a function that yields a value of subtype *CT*.Complex), also belongs to a *result interval*. The result intervals for the components of the result are either defined by a *maximum relative error* bound or by a *maximum box error* bound. When the result interval for the real (resp., imaginary) component is defined by maximum relative error, it is defined as for that of a real function, relative to the exact value of the real (resp., imaginary) part of the result of the corresponding mathematical function. When defined by maximum box error, the result interval for a component of the result is the smallest model interval of *CT*.Real that contains all the values of the corresponding part of $f \cdot (1.0 + d)$, where f is the exact complex value of the corresponding mathematical function at the given parameter values, d is complex, and $|d|$ is less than or equal to the given maximum box error. The function delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) when both bounds of the result interval(s) belong to the safe range of *CT*.Real; otherwise, 3/5

4 • if *CT*.Real'Machine_Overflows is True, the function either delivers a value that belongs to the result interval (or a value both of whose components belong to their respective result intervals) or raises Constraint_Error, signaling overflow;

5 • if *CT*.Real'Machine_Overflows is False, the result is implementation defined.

6/2 The error bounds for particular complex functions are tabulated in Table G.2. In the table, the error bound is given as the coefficient of *CT*.Real'Model_Epsilon.

7/1 *This paragraph was deleted.*

Table G.2: Error Bounds for Particular Complex Functions

Function or Operator	Nature of Result	Nature of Bound	Error Bound
Modulus	real	max. rel. error	3.0
Argument	real	max. rel. error	4.0
Compose_From_Polar	complex	max. rel. error	3.0
"*" (both operands complex)	complex	max. box error	5.0
"/" (right operand complex)	complex	max. box error	13.0
Sqrt	complex	max. rel. error	6.0
Log	complex	max. box error	13.0
Exp (complex parameter)	complex	max. rel. error	7.0
Exp (imaginary parameter)	complex	max. rel. error	2.0
Sin, Cos, Sinh, and Cosh	complex	max. rel. error	11.0
Tan, Cot, Tanh, and Coth	complex	max. rel. error	35.0
inverse trigonometric	complex	max. rel. error	14.0
inverse hyperbolic	complex	max. rel. error	14.0

8 The maximum relative error given above applies throughout the domain of the Compose_From_Polar function when the Cycle parameter is specified. When the Cycle parameter is omitted, the maximum relative error applies only when the absolute value of the parameter Argument is less than or equal to the angle threshold (see G.2.4). For the Exp function, and for the forward hyperbolic (resp., trigonometric) functions, the maximum relative error given above likewise applies only when the absolute value of the imaginary (resp., real) component of the parameter X (or the absolute value of the parameter itself, in the case of the Exp function with a parameter of pure-imaginary type) is less than or equal to the angle threshold. For larger angles, the accuracy is implementation defined.

9 The prescribed results specified in G.1.2 for certain functions at particular parameter values take precedence over the error bounds; effectively, they narrow to a single value the result interval allowed by the error bounds for a component of the result. Additional rules with a similar effect are given below for certain inverse trigonometric and inverse hyperbolic functions, at particular parameter values for which a component of the mathematical result is transcendental. In each case, the accuracy rule, which takes precedence over the error bounds, is that the result interval for the stated result component is the model interval of *CT*.Real associated with the component's exact mathematical value. The cases in question are as follows:

- When the parameter X has the value zero, the real (resp., imaginary) component of the result of the Arccot (resp., Arccoth) function is in the model interval of *CT*.Real associated with the value π/2.0. 10
- When the parameter X has the value one, the real component of the result of the Arcsin function is in the model interval of *CT*.Real associated with the value π/2.0. 11
- When the parameter X has the value –1.0, the real component of the result of the Arcsin (resp., Arccos) function is in the model interval of *CT*.Real associated with the value –π/2.0 (resp., π). 12

The amount by which a component of the result of an inverse trigonometric or inverse hyperbolic function is allowed to spill over into a quadrant adjacent to the one corresponding to the principal branch, as given in G.1.2, is limited. The rule is that the result belongs to the smallest model interval of *CT*.Real that contains both boundaries of the quadrant corresponding to the principal branch. This rule also takes precedence over the maximum error bounds, effectively narrowing the result interval allowed by them. 13/2

Finally, the results allowed by the error bounds are narrowed by one further rule: The absolute value of each component of the result of the Exp function, for a pure-imaginary parameter, never exceeds one. 14

Implementation Advice

The version of the Compose_From_Polar function without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter of 2.0*Numerics.Pi, since this will not provide the required accuracy in some portions of the domain. 15

G.3 Vector and Matrix Manipulation

Types and operations for the manipulation of real vectors and matrices are provided in Generic_Real_Arrays, which is defined in G.3.1. Types and operations for the manipulation of complex vectors and matrices are provided in Generic_Complex_Arrays, which is defined in G.3.2. Both of these library units are generic children of the predefined package Numerics (see A.5). Nongeneric equivalents of these packages for each of the predefined floating point types are also provided as children of Numerics. 1/2

G.3.1 Real Vectors and Matrices

Static Semantics

The generic library package Numerics.Generic_Real_Arrays has the following declaration: 1/2

```
generic                                                                    2/5
   type Real is digits <>;
package Ada.Numerics.Generic_Real_Arrays
   with Pure, Nonblocking is

   -- Types                                                               3/2

   type Real_Vector is array (Integer range <>) of Real'Base;              4/2
   type Real_Matrix is array (Integer range <>, Integer range <>)
                                                 of Real'Base;

   -- Subprograms for Real_Vector types                                   5/2

   -- Real_Vector arithmetic operations                                   6/2

   function "+"   (Right : Real_Vector)       return Real_Vector;         7/2
   function "-"   (Right : Real_Vector)       return Real_Vector;
   function "abs" (Right : Real_Vector)       return Real_Vector;

   function "+"   (Left, Right : Real_Vector) return Real_Vector;         8/2
   function "-"   (Left, Right : Real_Vector) return Real_Vector;

   function "*"   (Left, Right : Real_Vector) return Real'Base;           9/2
```

```
10/2        function "abs" (Right : Real_Vector)       return Real'Base;
11/2        -- Real_Vector scaling operations
12/2        function "*" (Left : Real'Base;   Right : Real_Vector)
               return Real_Vector;
            function "*" (Left : Real_Vector; Right : Real'Base)
               return Real_Vector;
            function "/" (Left : Real_Vector; Right : Real'Base)
               return Real_Vector;
13/2        -- Other Real_Vector operations
14/2        function Unit_Vector (Index : Integer;
                                  Order : Positive;
                                  First : Integer := 1) return Real_Vector;
15/2        -- Subprograms for Real_Matrix types
16/2        -- Real_Matrix arithmetic operations
17/2        function "+"       (Right : Real_Matrix) return Real_Matrix;
            function "-"       (Right : Real_Matrix) return Real_Matrix;
            function "abs"     (Right : Real_Matrix) return Real_Matrix;
            function Transpose (X     : Real_Matrix) return Real_Matrix;
18/2        function "+" (Left, Right : Real_Matrix) return Real_Matrix;
            function "-" (Left, Right : Real_Matrix) return Real_Matrix;
            function "*" (Left, Right : Real_Matrix) return Real_Matrix;
19/2        function "*" (Left, Right : Real_Vector) return Real_Matrix;
20/2        function "*" (Left : Real_Vector; Right : Real_Matrix)
               return Real_Vector;
            function "*" (Left : Real_Matrix; Right : Real_Vector)
               return Real_Vector;
21/2        -- Real_Matrix scaling operations
22/2        function "*" (Left : Real'Base;   Right : Real_Matrix)
               return Real_Matrix;
            function "*" (Left : Real_Matrix; Right : Real'Base)
               return Real_Matrix;
            function "/" (Left : Real_Matrix; Right : Real'Base)
               return Real_Matrix;
23/2        -- Real_Matrix inversion and related operations
24/2        function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;
            function Solve (A, X : Real_Matrix) return Real_Matrix;
            function Inverse (A : Real_Matrix) return Real_Matrix;
            function Determinant (A : Real_Matrix) return Real'Base;
25/2        -- Eigenvalues and vectors of a real symmetric matrix
26/2        function Eigenvalues (A : Real_Matrix) return Real_Vector;
27/2        procedure Eigensystem (A       : in  Real_Matrix;
                                   Values  : out Real_Vector;
                                   Vectors : out Real_Matrix);
28/2        -- Other Real_Matrix operations
29/2        function Unit_Matrix (Order            : Positive;
                                  First_1, First_2 : Integer := 1)
                                    return Real_Matrix;
30/2     end Ada.Numerics.Generic_Real_Arrays;
```

31/2 The library package Numerics.Real_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Real_Arrays, except that the predefined type Float is systematically substituted for Real'Base throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Real_Arrays, Numerics.Long_Real_Arrays, etc.

32/2 Two types are defined and exported by Numerics.Generic_Real_Arrays. The composite type Real_Vector is provided to represent a vector with components of type Real; it is defined as an unconstrained, one-

dimensional array with an index of type Integer. The composite type Real_Matrix is provided to represent a matrix with components of type Real; it is defined as an unconstrained, two-dimensional array with indices of type Integer.

The effect of the various subprograms is as described below. In most cases the subprograms are described in terms of corresponding scalar operations of the type Real; any exception raised by those operations is propagated by the array operation. Moreover, the accuracy of the result for each individual component is as defined for the scalar operation unless stated otherwise. 33/2

In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result is outside the range of Real'Base even though the mathematical final result would not be. 34/2

```
function "+"   (Right : Real_Vector) return Real_Vector;
function "-"   (Right : Real_Vector) return Real_Vector;
function "abs" (Right : Real_Vector) return Real_Vector;
```
35/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index range of the result is Right'Range. 36/2

```
function "+" (Left, Right : Real_Vector) return Real_Vector;
function "-" (Left, Right : Real_Vector) return Real_Vector;
```
37/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length. 38/2

```
function "*" (Left, Right : Real_Vector) return Real'Base;
```
39/2

This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product. 40/2

```
function "abs" (Right : Real_Vector) return Real'Base;
```
41/2

This operation returns the L2-norm of Right (the square root of the inner product of the vector with itself). 42/2

```
function "*" (Left : Real'Base; Right : Real_Vector) return Real_Vector;
```
43/2

This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index range of the result is Right'Range. 44/2

```
function "*" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
function "/" (Left : Real_Vector; Right : Real'Base) return Real_Vector;
```
45/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index range of the result is Left'Range. 46/2

```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Real_Vector;
```
47/2

This function returns a *unit vector* with Order components and a lower bound of First. All components are set to 0.0 except for the Index component which is set to 1.0. Constraint_Error is raised if Index < First, Index > First + Order – 1 or if First + Order – 1 > Integer'Last. 48/2

```
function "+"   (Right : Real_Matrix) return Real_Matrix;
function "-"   (Right : Real_Matrix) return Real_Matrix;
function "abs" (Right : Real_Matrix) return Real_Matrix;
```
49/2

Each operation returns the result of applying the corresponding operation of the type Real to each component of Right. The index ranges of the result are those of Right. 50/2

51/2 `function Transpose (X : Real_Matrix) return Real_Matrix;`

52/2 This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively.

53/2 `function "+" (Left, Right : Real_Matrix) return Real_Matrix;`
`function "-" (Left, Right : Real_Matrix) return Real_Matrix;`

54/2 Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2).

55/2 `function "*" (Left, Right : Real_Matrix) return Real_Matrix;`

56/2 This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products.

57/2 `function "*" (Left, Right : Real_Vector) return Real_Matrix;`

58/2 This operation returns the outer product of a (column) vector Left by a (row) vector Right using the operation "*" of the type Real for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

59/2 `function "*" (Left : Real_Vector; Right : Real_Matrix) return Real_Vector;`

60/2 This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products.

61/2 `function "*" (Left : Real_Matrix; Right : Real_Vector) return Real_Vector;`

62/2 This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products.

63/2 `function "*" (Left : Real'Base; Right : Real_Matrix) return Real_Matrix;`

64/2 This operation returns the result of multiplying each component of Right by the scalar Left using the "*" operation of the type Real. The index ranges of the result are those of Right.

65/2 `function "*" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;`
`function "/" (Left : Real_Matrix; Right : Real'Base) return Real_Matrix;`

66/2 Each operation returns the result of applying the corresponding operation of the type Real to each component of Left and to the scalar Right. The index ranges of the result are those of Left.

67/2 `function Solve (A : Real_Matrix; X : Real_Vector) return Real_Vector;`

68/2 This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

69/2 `function Solve (A, X : Real_Matrix) return Real_Matrix;`

70/2 This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the

result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned.

```
function Inverse (A : Real_Matrix) return Real_Matrix;
```

71/2

This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned.

72/2

```
function Determinant (A : Real_Matrix) return Real'Base;
```

73/2

This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2).

74/2

```
function Eigenvalues(A : Real_Matrix) return Real_Vector;
```

75/2

This function returns the eigenvalues of the symmetric matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not symmetric.

76/2

```
procedure Eigensystem(A       : in  Real_Matrix;
                      Values  : out Real_Vector;
                      Vectors : out Real_Matrix);
```

77/2

This procedure computes both the eigenvalues and eigenvectors of the symmetric matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are normalized and mutually orthogonal (they are orthonormal), including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2), or if Values'Range is not equal to A'Range(1), or if the index ranges of the parameter Vectors are not equal to those of A. Argument_Error is raised if the matrix A is not symmetric. Constraint_Error is also raised in implementation-defined circumstances if the algorithm used does not converge quickly enough.

78/3

```
function Unit_Matrix (Order            : Positive;
                      First_1, First_2 : Integer := 1) return Real_Matrix;
```

79/2

This function returns a square *unit matrix* with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to 0.0 except for the main diagonal, whose components are set to 1.0. Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last.

80/2

Implementation Requirements

Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

81/2

For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real in both the strict mode and the relaxed mode (see G.2).

82/2

For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g****abs**(*X*)***abs**(*Y*) where *g* is defined as

83/2

g = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)

84/2

For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* is defined as above.

85/2

Documentation Requirements

86/2 Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

Implementation Permissions

87/5 The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

Implementation Advice

88/3 Implementations should implement the Solve and Inverse functions using established techniques such as LU decomposition with row interchanges followed by back and forward substitution. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done, then it should be documented.

89/2 It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient.

90/2 The test that a matrix is symmetric should be performed by using the equality operator to compare the relevant components.

91/3 An implementation should minimize the circumstances under which the algorithm used for Eigenvalues and Eigensystem fails to converge.

G.3.2 Complex Vectors and Matrices

Static Semantics

1/2 The generic library package Numerics.Generic_Complex_Arrays has the following declaration:

2/5
```
with Ada.Numerics.Generic_Real_Arrays, Ada.Numerics.Generic_Complex_Types;
generic
   with package Real_Arrays   is new
      Ada.Numerics.Generic_Real_Arrays   (<>);
   use Real_Arrays;
   with package Complex_Types is new
      Ada.Numerics.Generic_Complex_Types (Real);
   use Complex_Types;
package Ada.Numerics.Generic_Complex_Arrays
   with Pure, Nonblocking is
```

3/2
```
   -- Types
```

4/2
```
   type Complex_Vector is array (Integer range <>) of Complex;
   type Complex_Matrix is array (Integer range <>,
                                 Integer range <>) of Complex;
```

5/2
```
   -- Subprograms for Complex_Vector types
```

6/2
```
   -- Complex_Vector selection, conversion and composition operations
```

7/2
```
   function Re (X : Complex_Vector) return Real_Vector;
   function Im (X : Complex_Vector) return Real_Vector;
```

8/2
```
   procedure Set_Re (X  : in out Complex_Vector;
                     Re : in     Real_Vector);
   procedure Set_Im (X  : in out Complex_Vector;
                     Im : in     Real_Vector);
```

9/2
```
   function Compose_From_Cartesian (Re     : Real_Vector)
      return Complex_Vector;
   function Compose_From_Cartesian (Re, Im : Real_Vector)
      return Complex_Vector;
```

```
function Modulus  (X     : Complex_Vector) return Real_Vector;                    10/2
function "abs"    (Right : Complex_Vector) return Real_Vector
                                              renames Modulus;
function Argument (X     : Complex_Vector) return Real_Vector;
function Argument (X     : Complex_Vector;
                   Cycle : Real'Base)      return Real_Vector;

function Compose_From_Polar (Modulus, Argument : Real_Vector)                     11/2
   return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                             Cycle             : Real'Base)
   return Complex_Vector;

-- Complex_Vector arithmetic operations                                           12/2

function "+"       (Right  : Complex_Vector) return Complex_Vector;               13/2
function "-"       (Right  : Complex_Vector) return Complex_Vector;
function Conjugate (X      : Complex_Vector) return Complex_Vector;

function "+"  (Left, Right : Complex_Vector) return Complex_Vector;               14/2
function "-"  (Left, Right : Complex_Vector) return Complex_Vector;

function "*"  (Left, Right : Complex_Vector) return Complex;                      15/2

function "abs"     (Right : Complex_Vector) return Real'Base;                     16/3

-- Mixed Real_Vector and Complex_Vector arithmetic operations                     17/2

function "+" (Left  : Real_Vector;                                                18/2
              Right : Complex_Vector) return Complex_Vector;
function "+" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;

function "*" (Left  : Real_Vector;    Right : Complex_Vector)                     19/2
   return Complex;
function "*" (Left  : Complex_Vector; Right : Real_Vector)
   return Complex;

-- Complex_Vector scaling operations                                              20/2

function "*" (Left  : Complex;                                                    21/2
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Complex)        return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Complex)        return Complex_Vector;

function "*" (Left  : Real'Base;                                                  22/2
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;
function "/" (Left  : Complex_Vector;
              Right : Real'Base)      return Complex_Vector;

-- Other Complex_Vector operations                                                23/2

function Unit_Vector (Index : Integer;                                            24/2
                      Order : Positive;
                      First : Integer := 1) return Complex_Vector;

-- Subprograms for Complex_Matrix types                                           25/2

-- Complex_Matrix selection, conversion and composition operations                26/2

function Re (X : Complex_Matrix) return Real_Matrix;                              27/2
function Im (X : Complex_Matrix) return Real_Matrix;

procedure Set_Re (X  : in out Complex_Matrix;                                     28/2
                  Re : in     Real_Matrix);
procedure Set_Im (X  : in out Complex_Matrix;
                  Im : in     Real_Matrix);
```

```
function Compose_From_Cartesian (Re     : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
   return Complex_Matrix;

function Modulus  (X     : Complex_Matrix) return Real_Matrix;
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                              renames Modulus;

function Argument (X     : Complex_Matrix) return Real_Matrix;
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;

function Compose_From_Polar (Modulus, Argument : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;

-- Complex_Matrix arithmetic operations

function "+"       (Right : Complex_Matrix) return Complex_Matrix;
function "-"       (Right : Complex_Matrix) return Complex_Matrix;
function Conjugate (X     : Complex_Matrix) return Complex_Matrix;
function Transpose (X     : Complex_Matrix) return Complex_Matrix;

function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;

function "*" (Left, Right : Complex_Vector) return Complex_Matrix;

function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;

-- Mixed Real_Matrix and Complex_Matrix arithmetic operations

function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;

function "*" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Matrix;
function "*" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Matrix;

function "*" (Left  : Real_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real_Matrix)    return Complex_Vector;
function "*" (Left  : Real_Matrix;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Real_Vector)    return Complex_Vector;

-- Complex_Matrix scaling operations

function "*" (Left  : Complex;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Complex)        return Complex_Matrix;
function "/" (Left  : Complex_Matrix;
              Right : Complex)        return Complex_Matrix;
```

```
   function "*" (Left  : Real'Base;                                          44/2
                 Right : Complex_Matrix) return Complex_Matrix;
   function "*" (Left  : Complex_Matrix;
                 Right : Real'Base)      return Complex_Matrix;
   function "/" (Left  : Complex_Matrix;
                 Right : Real'Base)      return Complex_Matrix;
   -- Complex_Matrix inversion and related operations                        45/2
   function Solve (A : Complex_Matrix; X : Complex_Vector)                   46/2
      return Complex_Vector;
   function Solve (A, X : Complex_Matrix) return Complex_Matrix;
   function Inverse (A : Complex_Matrix) return Complex_Matrix;
   function Determinant (A : Complex_Matrix) return Complex;
   -- Eigenvalues and vectors of a Hermitian matrix                          47/2
   function Eigenvalues(A : Complex_Matrix) return Real_Vector;              48/2
   procedure Eigensystem(A       : in  Complex_Matrix;                       49/2
                         Values  : out Real_Vector;
                         Vectors : out Complex_Matrix);
   -- Other Complex_Matrix operations                                        50/2
   function Unit_Matrix (Order            : Positive;                        51/2
                         First_1, First_2 : Integer := 1)
                                            return Complex_Matrix;
end Ada.Numerics.Generic_Complex_Arrays;                                     52/2
```

The library package Numerics.Complex_Arrays is declared pure and defines the same types and subprograms as Numerics.Generic_Complex_Arrays, except that the predefined type Float is systematically substituted for Real'Base, and the Real_Vector and Real_Matrix types exported by Numerics.Real_Arrays are systematically substituted for Real_Vector and Real_Matrix, and the Complex type exported by Numerics.Complex_Types is systematically substituted for Complex, throughout. Nongeneric equivalents for each of the other predefined floating point types are defined similarly, with the names Numerics.Short_Complex_Arrays, Numerics.Long_Complex_Arrays, etc. 53/2

Two types are defined and exported by Numerics.Generic_Complex_Arrays. The composite type Complex_Vector is provided to represent a vector with components of type Complex; it is defined as an unconstrained one-dimensional array with an index of type Integer. The composite type Complex_Matrix is provided to represent a matrix with components of type Complex; it is defined as an unconstrained, two-dimensional array with indices of type Integer. 54/2

The effect of the various subprograms is as described below. In many cases they are described in terms of corresponding scalar operations in Numerics.Generic_Complex_Types. Any exception raised by those operations is propagated by the array subprogram. Moreover, any constraints on the parameters and the accuracy of the result for each individual component are as defined for the scalar operation. 55/2

In the case of those operations which are defined to *involve an inner product*, Constraint_Error may be raised if an intermediate result has a component outside the range of Real'Base even though the final mathematical result would not. 56/2

```
function Re (X : Complex_Vector) return Real_Vector;                         57/2
function Im (X : Complex_Vector) return Real_Vector;
```

Each function returns a vector of the specified Cartesian components of X. The index range of the result is X'Range. 58/2

```
procedure Set_Re (X  : in out Complex_Vector; Re : in Real_Vector);          59/2
procedure Set_Im (X  : in out Complex_Vector; Im : in Real_Vector);
```

Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of 60/2

the components is unchanged. Constraint_Error is raised if X'Length is not equal to Re'Length or Im'Length.

61/2
```
function Compose_From_Cartesian (Re     : Real_Vector)
   return Complex_Vector;
function Compose_From_Cartesian (Re, Im : Real_Vector)
   return Complex_Vector;
```

62/2 Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index range of the result is Re'Range. Constraint_Error is raised if Re'Length is not equal to Im'Length.

63/2
```
function Modulus  (X     : Complex_Vector) return Real_Vector;
function "abs"    (Right : Complex_Vector) return Real_Vector
                                              renames Modulus;
function Argument (X     : Complex_Vector) return Real_Vector;
function Argument (X     : Complex_Vector;
                   Cycle : Real'Base)      return Real_Vector;
```

64/2 Each function calculates and returns a vector of the specified polar components of X or Right using the corresponding function in numerics.generic_complex_types. The index range of the result is X'Range or Right'Range.

65/2
```
function Compose_From_Polar (Modulus, Argument : Real_Vector)
   return Complex_Vector;
function Compose_From_Polar (Modulus, Argument : Real_Vector;
                             Cycle             : Real'Base)
   return Complex_Vector;
```

66/2 Each function constructs a vector of Complex results (in Cartesian representation) formed from given vectors of polar components using the corresponding function in numerics.-generic_complex_types on matching components of Modulus and Argument. The index range of the result is Modulus'Range. Constraint_Error is raised if Modulus'Length is not equal to Argument'Length.

67/2
```
function "+" (Right : Complex_Vector) return Complex_Vector;
function "-" (Right : Complex_Vector) return Complex_Vector;
```

68/2 Each operation returns the result of applying the corresponding operation in numerics.-generic_complex_types to each component of Right. The index range of the result is Right'Range.

69/2
```
function Conjugate (X : Complex_Vector) return Complex_Vector;
```

70/2 This function returns the result of applying the appropriate function Conjugate in numerics.-generic_complex_types to each component of X. The index range of the result is X'Range.

71/2
```
function "+" (Left, Right : Complex_Vector) return Complex_Vector;
function "-" (Left, Right : Complex_Vector) return Complex_Vector;
```

72/2 Each operation returns the result of applying the corresponding operation in numerics.-generic_complex_types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length.

73/2
```
function "*" (Left, Right : Complex_Vector) return Complex;
```

74/2 This operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. This operation involves an inner product.

```
function "abs" (Right : Complex_Vector) return Real'Base;
```
75/3

This operation returns the Hermitian L2-norm of Right (the square root of the inner product of the vector with its conjugate). 76/2

```
function "+" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "+" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
function "-" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Vector;
function "-" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Vector;
```
77/2

Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index range of the result is Left'Range. Constraint_Error is raised if Left'Length is not equal to Right'Length. 78/2

```
function "*" (Left : Real_Vector;    Right : Complex_Vector) return Complex;
function "*" (Left : Complex_Vector; Right : Real_Vector)    return Complex;
```
79/2

Each operation returns the inner product of Left and Right. Constraint_Error is raised if Left'Length is not equal to Right'Length. These operations involve an inner product. 80/2

```
function "*" (Left : Complex; Right : Complex_Vector) return Complex_Vector;
```
81/2

This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in numerics.generic_complex_types. The index range of the result is Right'Range. 82/2

```
function "*" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
function "/" (Left : Complex_Vector; Right : Complex) return Complex_Vector;
```
83/2

Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the vector Left and the complex number Right. The index range of the result is Left'Range. 84/2

```
function "*" (Left : Real'Base;
              Right : Complex_Vector) return Complex_Vector;
```
85/2

This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in numerics.generic_complex_types. The index range of the result is Right'Range. 86/2

```
function "*" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
function "/" (Left : Complex_Vector;
              Right : Real'Base) return Complex_Vector;
```
87/2

Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of the vector Left and the real number Right. The index range of the result is Left'Range. 88/2

```
function Unit_Vector (Index : Integer;
                      Order : Positive;
                      First : Integer := 1) return Complex_Vector;
```
89/2

This function returns a *unit vector* with Order components and a lower bound of First. All components are set to (0.0, 0.0) except for the Index component which is set to (1.0, 0.0). Constraint_Error is raised if Index < First, Index > First + Order – 1, or if First + Order – 1 > Integer'Last. 90/2

91/2
```
function Re (X : Complex_Matrix) return Real_Matrix;
function Im (X : Complex_Matrix) return Real_Matrix;
```

92/2 Each function returns a matrix of the specified Cartesian components of X. The index ranges of the result are those of X.

93/2
```
procedure Set_Re (X : in out Complex_Matrix; Re : in Real_Matrix);
procedure Set_Im (X : in out Complex_Matrix; Im : in Real_Matrix);
```

94/2 Each procedure replaces the specified (Cartesian) component of each of the components of X by the value of the matching component of Re or Im; the other (Cartesian) component of each of the components is unchanged. Constraint_Error is raised if X'Length(1) is not equal to Re'Length(1) or Im'Length(1) or if X'Length(2) is not equal to Re'Length(2) or Im'Length(2).

95/2
```
function Compose_From_Cartesian (Re     : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Cartesian (Re, Im : Real_Matrix)
   return Complex_Matrix;
```

96/2 Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of Cartesian components; when only the real components are given, imaginary components of zero are assumed. The index ranges of the result are those of Re. Constraint_Error is raised if Re'Length(1) is not equal to Im'Length(1) or Re'Length(2) is not equal to Im'Length(2).

97/2
```
function Modulus  (X     : Complex_Matrix) return Real_Matrix;
function "abs"    (Right : Complex_Matrix) return Real_Matrix
                                              renames Modulus;
function Argument (X     : Complex_Matrix) return Real_Matrix;
function Argument (X     : Complex_Matrix;
                   Cycle : Real'Base)      return Real_Matrix;
```

98/2 Each function calculates and returns a matrix of the specified polar components of X or Right using the corresponding function in numerics.generic_complex_types. The index ranges of the result are those of X or Right.

99/2
```
function Compose_From_Polar (Modulus, Argument : Real_Matrix)
   return Complex_Matrix;
function Compose_From_Polar (Modulus, Argument : Real_Matrix;
                             Cycle             : Real'Base)
   return Complex_Matrix;
```

100/2 Each function constructs a matrix of Complex results (in Cartesian representation) formed from given matrices of polar components using the corresponding function in numerics.generic_complex_types on matching components of Modulus and Argument. The index ranges of the result are those of Modulus. Constraint_Error is raised if Modulus'Length(1) is not equal to Argument'Length(1) or Modulus'Length(2) is not equal to Argument'Length(2).

101/2
```
function "+" (Right : Complex_Matrix) return Complex_Matrix;
function "-" (Right : Complex_Matrix) return Complex_Matrix;
```

102/2 Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Right. The index ranges of the result are those of Right.

103/2
```
function Conjugate (X : Complex_Matrix) return Complex_Matrix;
```

104/2 This function returns the result of applying the appropriate function Conjugate in numerics.generic_complex_types to each component of X. The index ranges of the result are those of X.

```
function Transpose (X : Complex_Matrix) return Complex_Matrix;
```
105/2

This function returns the transpose of a matrix X. The first and second index ranges of the result are X'Range(2) and X'Range(1) respectively. 106/2

```
function "+" (Left, Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left, Right : Complex_Matrix) return Complex_Matrix;
```
107/2

Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2). 108/2

```
function "*" (Left, Right : Complex_Matrix) return Complex_Matrix;
```
109/2

This operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). This operation involves inner products. 110/2

```
function "*" (Left, Right : Complex_Vector) return Complex_Matrix;
```
111/2

This operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in numerics.generic_complex_types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively. 112/2

```
function "*" (Left  : Complex_Vector;
              Right : Complex_Matrix) return Complex_Vector;
```
113/2

This operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). This operation involves inner products. 114/2

```
function "*" (Left  : Complex_Matrix;
              Right : Complex_Vector) return Complex_Vector;
```
115/2

This operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. This operation involves inner products. 116/2

```
function "+" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "+" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
function "-" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "-" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```
117/2

Each operation returns the result of applying the corresponding operation in numerics.generic_complex_types to each component of Left and the matching component of Right. The index ranges of the result are those of Left. Constraint_Error is raised if Left'Length(1) is not equal to Right'Length(1) or Left'Length(2) is not equal to Right'Length(2). 118/2

119/2
```
function "*" (Left  : Real_Matrix;
              Right : Complex_Matrix) return Complex_Matrix;
function "*" (Left  : Complex_Matrix;
              Right : Real_Matrix)    return Complex_Matrix;
```

120/2 Each operation provides the standard mathematical operation for matrix multiplication. The first and second index ranges of the result are Left'Range(1) and Right'Range(2) respectively. Constraint_Error is raised if Left'Length(2) is not equal to Right'Length(1). These operations involve inner products.

121/2
```
function "*" (Left  : Real_Vector;
              Right : Complex_Vector) return Complex_Matrix;
function "*" (Left  : Complex_Vector;
              Right : Real_Vector)    return Complex_Matrix;
```

122/2 Each operation returns the outer product of a (column) vector Left by a (row) vector Right using the appropriate operation "*" in numerics.generic_complex_types for computing the individual components. The first and second index ranges of the result are Left'Range and Right'Range respectively.

123/2
```
function "*" (Left  : Real_Vector;
              Right : Complex_Matrix) return Complex_Vector;
function "*" (Left  : Complex_Vector;
              Right : Real_Matrix)    return Complex_Vector;
```

124/2 Each operation provides the standard mathematical operation for multiplication of a (row) vector Left by a matrix Right. The index range of the (row) vector result is Right'Range(2). Constraint_Error is raised if Left'Length is not equal to Right'Length(1). These operations involve inner products.

125/2
```
function "*" (Left  : Real_Matrix;
              Right : Complex_Vector) return Complex_Vector;
function "*" (Left  : Complex_Matrix;
              Right : Real_Vector)    return Complex_Vector;
```

126/2 Each operation provides the standard mathematical operation for multiplication of a matrix Left by a (column) vector Right. The index range of the (column) vector result is Left'Range(1). Constraint_Error is raised if Left'Length(2) is not equal to Right'Length. These operations involve inner products.

127/2
```
function "*" (Left : Complex; Right : Complex_Matrix) return Complex_Matrix;
```

128/2 This operation returns the result of multiplying each component of Right by the complex number Left using the appropriate operation "*" in numerics.generic_complex_types. The index ranges of the result are those of Right.

129/2
```
function "*" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
function "/" (Left : Complex_Matrix; Right : Complex) return Complex_Matrix;
```

130/2 Each operation returns the result of applying the corresponding operation in numerics.-generic_complex_types to each component of the matrix Left and the complex number Right. The index ranges of the result are those of Left.

131/2
```
function "*" (Left : Real'Base;
              Right : Complex_Matrix) return Complex_Matrix;
```

132/2 This operation returns the result of multiplying each component of Right by the real number Left using the appropriate operation "*" in numerics.generic_complex_types. The index ranges of the result are those of Right.

```
function "*" (Left : Complex_Matrix;                                    133/2
              Right : Real'Base) return Complex_Matrix;
function "/" (Left : Complex_Matrix;
              Right : Real'Base) return Complex_Matrix;
```

Each operation returns the result of applying the corresponding operation in numerics.-generic_complex_types to each component of the matrix Left and the real number Right. The index ranges of the result are those of Left. 134/2

```
function Solve (A : Complex_Matrix; X : Complex_Vector) return          135/2
Complex_Vector;
```

This function returns a vector Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving a single set of linear equations. The index range of the result is A'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length are not equal. Constraint_Error is raised if the matrix A is ill-conditioned. 136/2

```
function Solve (A, X : Complex_Matrix) return Complex_Matrix;           137/2
```

This function returns a matrix Y such that X is (nearly) equal to A * Y. This is the standard mathematical operation for solving several sets of linear equations. The index ranges of the result are A'Range(2) and X'Range(2). Constraint_Error is raised if A'Length(1), A'Length(2), and X'Length(1) are not equal. Constraint_Error is raised if the matrix A is ill-conditioned. 138/2

```
function Inverse (A : Complex_Matrix) return Complex_Matrix;            139/2
```

This function returns a matrix B such that A * B is (nearly) equal to the unit matrix. The index ranges of the result are A'Range(2) and A'Range(1). Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). Constraint_Error is raised if the matrix A is ill-conditioned. 140/2

```
function Determinant (A : Complex_Matrix) return Complex;               141/2
```

This function returns the determinant of the matrix A. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). 142/2

```
function Eigenvalues(A : Complex_Matrix) return Real_Vector;            143/2
```

This function returns the eigenvalues of the Hermitian matrix A as a vector sorted into order with the largest first. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2). The index range of the result is A'Range(1). Argument_Error is raised if the matrix A is not Hermitian. 144/2

```
procedure Eigensystem(A       : in  Complex_Matrix;                     145/2
                      Values  :  out Real_Vector;
                      Vectors :  out Complex_Matrix);
```

This procedure computes both the eigenvalues and eigenvectors of the Hermitian matrix A. The out parameter Values is the same as that obtained by calling the function Eigenvalues. The out parameter Vectors is a matrix whose columns are the eigenvectors of the matrix A. The order of the columns corresponds to the order of the eigenvalues. The eigenvectors are mutually orthonormal, including when there are repeated eigenvalues. Constraint_Error is raised if A'Length(1) is not equal to A'Length(2), or if Values'Range is not equal to A'Range(1), or if the index ranges of the parameter Vectors are not equal to those of A. Argument_Error is raised if the matrix A is not Hermitian. Constraint_Error is also raised in implementation-defined circumstances if the algorithm used does not converge quickly enough. 146/3

147/2
```
function Unit_Matrix (Order            : Positive;
                      First_1, First_2 : Integer := 1)
                                         return Complex_Matrix;
```

148/2 This function returns a square *unit matrix* with Order**2 components and lower bounds of First_1 and First_2 (for the first and second index ranges respectively). All components are set to (0.0, 0.0) except for the main diagonal, whose components are set to (1.0, 0.0). Constraint_Error is raised if First_1 + Order – 1 > Integer'Last or First_2 + Order – 1 > Integer'Last.

Implementation Requirements

149/2 Accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem are implementation defined.

150/2 For operations not involving an inner product, the accuracy requirements are those of the corresponding operations of the type Real'Base and Complex in both the strict mode and the relaxed mode (see G.2).

151/2 For operations involving an inner product, no requirements are specified in the relaxed mode. In the strict mode the modulus of the absolute error of the inner product *X***Y* shall not exceed *g****abs**(*X*)***abs**(*Y*) where *g* is defined as

152/2 *g* = *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)
for mixed complex and real operands

153/2 *g* = sqrt(2.0) * *X*'Length * Real'Machine_Radix**(1 – Real'Model_Mantissa)
for two complex operands

154/2 For the L2-norm, no accuracy requirements are specified in the relaxed mode. In the strict mode the relative error on the norm shall not exceed *g* / 2.0 + 3.0 * Real'Model_Epsilon where *g* has the definition appropriate for two complex operands.

Documentation Requirements

155/2 Implementations shall document any techniques used to reduce cancellation errors such as extended precision arithmetic.

Implementation Permissions

156/5 The nongeneric equivalent packages can be actual instantiations of the generic package for the appropriate predefined type, though that is not required.

157/5 Although many operations are defined in terms of operations from Numerics.Generic_Complex_Types, they can be implemented by other operations that have the same effect.

Implementation Advice

158/3 Implementations should implement the Solve and Inverse functions using established techniques. Implementations are recommended to refine the result by performing an iteration on the residuals; if this is done, then it should be documented.

159/2 It is not the intention that any special provision should be made to determine whether a matrix is ill-conditioned or not. The naturally occurring overflow (including division by zero) which will result from executing these functions with an ill-conditioned matrix and thus raise Constraint_Error is sufficient.

160/2 The test that a matrix is Hermitian should use the equality operator to compare the real components and negation followed by equality to compare the imaginary components (see G.2.1).

An implementation should minimize the circumstances under which the algorithm used for Eigenvalues and Eigensystem fails to converge. 160.1/3

Implementations should not perform operations on mixed complex and real operands by first converting the real operand to complex. See G.1.1. 161/2

Annex H
(normative)
High Integrity Systems

This Annex addresses requirements for high integrity systems (including safety-critical systems and security-critical systems). It provides facilities and specifies documentation requirements that relate to several needs: 1/2

- Understanding program execution; 2
- Reviewing object code; 3
- Restricting language constructs whose usage can complicate the demonstration of program correctness 4/5

Execution understandability is supported by pragma Normalize_Scalars, and also by requirements for the implementation to document the effect of a program in the presence of a bounded error or where the language rules leave the effect unspecified. 4.1

The pragmas Reviewable and Restrictions relate to the other requirements addressed by this Annex. 5

NOTE The Valid attribute (see 13.9.2) is also useful in addressing these needs, to avoid problems that can otherwise arise from scalars that have values outside their declared range constraints. 6/5

H.1 Pragma Normalize_Scalars

This pragma ensures that an otherwise uninitialized scalar object is set to a predictable value, but out of range if possible. 1

Syntax

The form of a pragma Normalize_Scalars is as follows: 2

pragma Normalize_Scalars; 3

Post-Compilation Rules

Pragma Normalize_Scalars is a configuration pragma. It applies to all compilation_units included in a partition. 4

Documentation Requirements

If a pragma Normalize_Scalars applies, the implementation shall document the implicit initial values for scalar subtypes, and shall identify each case in which such a value is used and is not an invalid representation. 5/2

Implementation Advice

Whenever possible, the implicit initial values for a scalar subtype should be an invalid representation (see 13.9.1). 6/2

NOTE 1 The initialization requirement applies to uninitialized scalar objects that are subcomponents of composite objects, to allocated objects, and to stand-alone objects. It also applies to scalar **out** parameters. Scalar subcomponents of composite **out** parameters are initialized to the corresponding part of the actual, by virtue of 6.4.1. 7

NOTE 2 The initialization requirement does not apply to a scalar for which pragma Import has been specified, since initialization of an imported object is performed solely by the foreign language environment (see B.1). 8

9/5 NOTE 3 The use of pragma Normalize_Scalars in conjunction with Pragma Restrictions(No_Exceptions) can result in erroneous execution (see H.4).

H.2 Documentation of Implementation Decisions

Documentation Requirements

1/5 The implementation shall document the range of effects for each situation that the language rules identify as either a bounded error or as having an unspecified effect. If the implementation can constrain the effects of erroneous execution for a given construct, then it shall document such constraints. The documentation may be provided either independently of any compilation unit or partition, or as part of an annotated listing for a given unit or partition. See also 1.1.3, and 1.1.2.

2 NOTE Among the situations to be documented are the conventions chosen for parameter passing, the methods used for the management of run-time storage, and the method used to evaluate numeric expressions if this involves extended range or extra precision.

H.3 Reviewable Object Code

1 Object code review and validation are supported by pragmas Reviewable and Inspection_Point.

H.3.1 Pragma Reviewable

1 This pragma directs the implementation to provide information to facilitate analysis and review of a program's object code, in particular to allow determination of execution time and storage usage and to identify the correspondence between the source and object programs.

Syntax

2 The form of a pragma Reviewable is as follows:

3 **pragma** Reviewable;

Post-Compilation Rules

4 Pragma Reviewable is a configuration pragma. It applies to all compilation_units included in a partition.

Implementation Requirements

5 The implementation shall provide the following information for any compilation unit to which such a pragma applies:

6 • Where compiler-generated runtime checks remain;

7 • An identification of any construct with a language-defined check that is recognized prior to run time as certain to fail if executed (even if the generation of runtime checks has been suppressed);

8/2 • For each read of a scalar object, an identification of the read as either “known to be initialized”, or “possibly uninitialized”, independent of whether pragma Normalize_Scalars applies;

9 • Where run-time support routines are implicitly invoked;

10 • An object code listing, including:

11 • Machine instructions, with relative offsets;

12 • Where each data object is stored during its lifetime;

- Correspondence with the source program, including an identification of the code produced per declaration and per statement. 13

- An identification of each construct for which the implementation detects the possibility of erroneous execution; 14

- For each subprogram, block, task, or other construct implemented by reserving and subsequently freeing an area on a run-time stack, an identification of the length of the fixed-size portion of the area and an indication of whether the non-fixed size portion is reserved on the stack or in a dynamically-managed storage region. 15

The implementation shall provide the following information for any partition to which the pragma applies: 16

- An object code listing of the entire partition, including initialization and finalization code as well as run-time system components, and with an identification of those instructions and data that will be relocated at load time; 17

- A description of the run-time model relevant to the partition. 18

The implementation shall provide control- and data-flow information, both within each compilation unit and across the compilation units of the partition. 18.1

Implementation Advice

The implementation should provide the above information in both a human-readable and machine-readable form, and should document the latter so as to ease further processing by automated tools. 19

Object code listings should be provided both in a symbolic format and also in an appropriate numeric format (such as hexadecimal or octal). 20

NOTE The order of elaboration of library units will be documented even in the absence of pragma Reviewable (see 10.2). 21

H.3.2 Pragma Inspection_Point

An occurrence of a pragma Inspection_Point identifies a set of objects each of whose values is to be available at the point(s) during program execution corresponding to the position of the pragma in the compilation unit. The purpose of such a pragma is to facilitate code validation. 1

Syntax

The form of a pragma Inspection_Point is as follows: 2

pragma Inspection_Point[(*object*_name {, *object*_name})]; 3

Legality Rules

A pragma Inspection_Point is an executable pragma (see 2.8), and is allowed wherever a basic_declaration, declare_item, or statement is allowed. Each *object*_name shall statically denote the declaration of an object. 4/6

Static Semantics

An *inspection point* is a point in the object code corresponding to the occurrence of a pragma Inspection_Point in the compilation unit. An object is *inspectable* at an inspection point if the corresponding pragma Inspection_Point either has an argument denoting that object, or has no arguments and the declaration of the object is visible at the inspection point. 5/2

Dynamic Semantics

6 Execution of a pragma Inspection_Point has no effect.

Implementation Requirements

7 Reaching an inspection point is an external interaction with respect to the values of the inspectable objects at that point (see 1.1.3).

Documentation Requirements

8 For each inspection point, the implementation shall identify a mapping between each inspectable object and the machine resources (such as memory locations or registers) from which the object's value can be obtained.

9/5 NOTE 1 Because reaching an inspection point is considered an external interaction relative to the values of the inspectable variables, the implementation cannot perform “dead store elimination” on the last assignment to such a variable prior to an inspection point. Thus an inspection point has the effect of an implicit read of each of its inspectable objects.

10 NOTE 2 Inspection points are useful in maintaining a correspondence between the state of the program in source code terms, and the machine state during the program's execution. Assertions about the values of program objects can be tested in machine terms at inspection points. Object code between inspection points can be processed by automated tools to verify programs mechanically.

11/5 NOTE 3 The identification of the mapping from source program objects to machine resources can be in the form of an annotated object listing, in human-readable or tool-processable form.

H.4 High Integrity Restrictions

1/3 This subclause defines restrictions that can be used with pragma Restrictions (see 13.12); these facilitate the demonstration of program correctness by allowing tailored versions of the run-time system.

Static Semantics

2/2 *This paragraph was deleted.*

3/2 The following *restriction*_identifiers are language defined:

4 **Tasking-related restriction:**

5 No_Protected_Types
: There are no declarations of protected types or protected objects.

6 **Memory-management related restrictions:**

7 No_Allocators
: There are no occurrences of an allocator.

8/1 No_Local_Allocators
: Allocators are prohibited in subprograms, generic subprograms, tasks, and entry bodies.

8.1/3 No_Anonymous_Allocators
: There are no allocators of anonymous access types.

8.2/3 No_Coextensions
: There are no coextensions. See 3.10.2.

8.3/3 No_Access_Parameter_Allocators
: Allocators are not permitted as the actual parameter to an access parameter. See 6.1.

9/2 *This paragraph was deleted.*

Immediate_Reclamation 10
Except for storage occupied by objects created by allocators and not deallocated via unchecked deallocation, any storage reserved at run time for an object is immediately reclaimed when the object no longer exists.

Exception-related restriction: 11

No_Exceptions 12/5
Raise_statements and exception_handlers are not allowed. No language-defined runtime checks are generated; however, a runtime check performed automatically by the hardware is permitted. The callable entity associated with a procedural_iterator (see 5.5.3) is considered to not allow exit, independent of the value of its Allows_Exit aspect.

Other restrictions: 13

No_Floating_Point 14
Uses of predefined floating point types and operations, and declarations of new floating point types, are not allowed.

No_Fixed_Point 15
Uses of predefined fixed point types and operations, and declarations of new fixed point types, are not allowed.

This paragraph was deleted. 16/2

No_Access_Subprograms 17
The declaration of access-to-subprogram types is not allowed.

No_Unchecked_Access 18
The Unchecked_Access attribute is not allowed.

No_Dispatch Occurrences of T'Class are not allowed, for any (tagged) subtype T. 19

No_IO Semantic dependence on any of the library units Sequential_IO, Direct_IO, Text_IO, Wide_Text_IO, Wide_Wide_Text_IO, Stream_IO, or Directories is not allowed. 20/5

No_Delay Delay_Statements and semantic dependence on package Calendar are not allowed. 21

No_Recursion 22
As part of the execution of a subprogram, the same subprogram is not invoked.

No_Reentrancy 23
During the execution of a subprogram by a task, no other task invokes the same subprogram.

No_Unspecified_Globals 23.1/5
No library-level entity shall have a Global aspect of Unspecified, either explicitly or by default. No library-level entity shall have a Global'Class aspect of Unspecified, explicitly or by default, if it is used as part of a dispatching call.

No_Hidden_Indirect_Globals 23.2/5
When within a context where an applicable global aspect is neither Unspecified nor **in out all**, any execution within such a context does neither of the following:

- Update (or return a writable reference to) a variable that is reachable via a sequence of zero or more dereferences of access-to-object values from a parameter of a visibly access-to-constant type, from a part of a non-access-type formal parameter of mode **in** (after any **overriding** – see H.7), or from a global that has mode **in** or is not within the applicable global variable set, unless the initial dereference is of a part of a formal parameter or global that is visibly of an access-to-variable type; 23.3/5

23.4/5 • Read (or return a readable reference to) a variable that is reachable via a sequence of zero or more dereferences of access-to-object values from a global that is not within the applicable global variable set, unless the initial dereference is of a part of a formal parameter or global that is visibly of an access-to-object type.

23.5/5 For the purposes of the above rules:

23.6/5 • a part of an object is *visibly of an access type* if the type of the object is declared immediately within the visible part of a package specification, and at the point of declaration of the type the part is visible and of an access type;

23.7/5 • a function *returns a writable reference to V* if it returns a result with a part that is visibly of an access-to-variable type designating *V*; similarly, a function *returns a readable reference to V* if it returns a result with a part that is visibly of an access-to-constant type designating *V*;

23.8/5 • if an applicable global variable set includes a package name, and the collection of some pool-specific access type (see 7.6.1) is implicitly declared in a part of the declarative region of the package included within the global variable set, then all objects allocated from that collection are considered included within the global variable set.

23.9/5 The consequences of violating the No_Hidden_Indirect_Globals restriction is implementation-defined. Any aspects or other means for identifying such violations prior to or during execution are implementation-defined.

Dynamic Semantics

23.10/5 The following *restriction_parameter*_identifier is language defined:

23.11/5 Max_Image_Length
Specifies the maximum length for the result of an Image, Wide_Image, or Wide_Wide_Image attribute. Violation of this restriction results in the raising of Program_Error at the point of the invocation of an image attribute.

Implementation Requirements

23.12/5 An implementation of this Annex shall support:

23.13/5 • the restrictions defined in this subclause; and

23.14/5 • the following restrictions defined in D.7: No_Task_Hierarchy, No_Abort_Statement, No_Implicit_Heap_Allocation, No_Standard_Allocators_After_Elaboration; and

23.15/5 • the **pragma** Profile(Ravenscar); and

23.16/5 • the following uses of *restriction_parameter*_identifiers defined in D.7, which are checked prior to program execution:

23.17/5 • Max_Task_Entries => 0,

23.18/5 • Max_Asynchronous_Select_Nesting => 0, and

23.19/5 • Max_Tasks => 0.

23.20/5 If a Max_Image_Length restriction applies to any compilation unit in the partition, then for any subtype S, S'Image, S'Wide_Image, and S'Wide_Wide_Image shall be implemented within that partition without any dynamic allocation.

24/5 If an implementation supports pragma Restrictions for a particular argument, then except for the restrictions No_Access_Subprograms, No_Unchecked_Access, No_Specification_of_Aspect, No_Use_of_Attribute, No_Use_of_Pragma, No_Dependence => Ada.Unchecked_Conversion, and

No_Dependence => Ada.Unchecked_Deallocation, the associated restriction applies to the run-time system.

Documentation Requirements

If a pragma Restrictions(No_Exceptions) is specified, the implementation shall document the effects of all constructs where language-defined checks are still performed automatically (for example, an overflow check performed by the processor). 25

Erroneous Execution

Program execution is erroneous if pragma Restrictions(No_Exceptions) has been specified and the conditions arise under which a generated language-defined runtime check would fail. 26

Program execution is erroneous if pragma Restrictions(No_Recursion) has been specified and a subprogram is invoked as part of its own execution, or if pragma Restrictions(No_Reentrancy) has been specified and during the execution of a subprogram by a task, another task invokes the same subprogram. 27

NOTE Uses of *restriction_parameter*_identifier No_Dependence defined in 13.12.1: No_Dependence => Ada.-Unchecked_Deallocation and No_Dependence => Ada.Unchecked_Conversion can be appropriate for high-integrity systems. Other uses of No_Dependence can also be appropriate for high-integrity systems. 28/5

H.4.1 Aspect No_Controlled_Parts

Static Semantics

For a type, the following type-related, operational aspect may be specified: 1/5

No_Controlled_Parts 2/5

The type of this aspect is Boolean. If True, the type and any descendants shall not have any controlled parts. If specified, the value of the expression shall be static. If not specified, the value of this aspect is False.

The No_Controlled_Parts aspect is nonoverridable (see 13.1.1). 3/5

Legality Rules

If No_Controlled_Parts is True for a type, no component of the type shall have a controlled part nor shall the type itself be controlled. For the purposes of this rule, a type has a controlled part if its full type has a controlled part; this is applied recursively. In addition to the places where Legality Rules normally apply (see 12.3), this rule also applies in the private part of an instance of a generic unit. 4/5

When enforcing the above rule within a generic body *G* or within the body of a generic unit declared within the declarative region of generic unit *G*, a generic formal private type of *G* and a generic formal derived type of *G* whose ancestor is a tagged type whose No_Controlled_Parts aspect is False are considered to have a controlled part. 5/5

H.5 Pragma Detect_Blocking

The following pragma requires an implementation to detect potentially blocking operations during the execution of a protected operation or a parallel construct. 1/5

Syntax

The form of a pragma Detect_Blocking is as follows: 2/2

pragma Detect_Blocking; 3/2

Post-Compilation Rules

4/2 A pragma Detect_Blocking is a configuration pragma.

Dynamic Semantics

5/5 An implementation is required to detect a potentially blocking operation that occurs during the execution of a protected operation or a parallel construct defined within a compilation unit to which the pragma applies, and to raise Program_Error (see 9.5).

Implementation Permissions

6/5 An implementation is allowed to reject a compilation_unit to which a pragma Detect_Blocking applies if a potentially blocking operation is present directly within an entry_body, the body of a protected subprogram, or a parallel construct occurring within the compilation unit.

7/5 NOTE An operation that causes a task to be blocked within a foreign language domain is not defined to be potentially blocking, and is unlikely to be detected.

H.6 Pragma Partition_Elaboration_Policy

1/3 This subclause defines a pragma for user control over elaboration policy.

Syntax

2/2 The form of a pragma Partition_Elaboration_Policy is as follows:

3/2 **pragma** Partition_Elaboration_Policy (*policy_*identifier);

4/2 The *policy_*identifier shall be either Sequential, Concurrent or an implementation-defined identifier.

Post-Compilation Rules

5/2 A pragma Partition_Elaboration_Policy is a configuration pragma. It specifies the elaboration policy for a partition. At most one elaboration policy shall be specified for a partition.

6/3 If the Sequential policy is specified for a partition, then pragma Restrictions (No_Task_Hierarchy) shall also be specified for the partition.

Dynamic Semantics

7/2 Notwithstanding what this document says elsewhere, this pragma allows partition elaboration rules concerning task activation and interrupt attachment to be changed. If the *policy_*identifier is Concurrent, or if there is no pragma Partition_Elaboration_Policy defined for the partition, then the rules defined elsewhere in this Reference Manual apply.

8/2 If the partition elaboration policy is Sequential, then task activation and interrupt attachment are performed in the following sequence of steps:

9/2 - The activation of all library-level tasks and the attachment of interrupt handlers are deferred until all library units are elaborated.

10/2 - The interrupt handlers are attached by the environment task.

11/2 - The environment task is suspended while the library-level tasks are activated.

12/2 - The environment task executes the main subprogram (if any) concurrently with these executing tasks.

If several dynamic interrupt handler attachments for the same interrupt are deferred, then the most recent call of Attach_Handler or Exchange_Handler determines which handler is attached. 13/2

If any deferred task activation fails, Tasking_Error is raised at the beginning of the sequence of statements of the body of the environment task prior to calling the main subprogram. 14/2

Implementation Advice

If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration, then the partition is deadlocked and it is recommended that the partition be immediately terminated. 15/3

Implementation Permissions

If the partition elaboration policy is Sequential and any task activation fails, then an implementation may immediately terminate the active partition to mitigate the hazard posed by continuing to execute with a subset of the tasks being active. 16/3

NOTE If any deferred task activation fails, the environment task is unable to handle the Tasking_Error exception and completes immediately. By contrast, if the partition elaboration policy is Concurrent, then this exception can be handled within a library unit. 17/5

H.7 Extensions to Global and Global'Class Aspects

In addition to the entities specified in 6.1.2, the Global aspect may be specified for a subtype (including a formal subtype), formal package, formal subprogram, and formal object of an anonymous access-to-subprogram type. 1/5

Syntax

The following additional syntax is provided to override the mode of a formal parameter to reflect indirect effects on variables reachable from the formal parameter by one or more access-value dereferences: 2/5

```
extended_global_mode ::=                                    3/5
   overriding basic_global_mode
```

Name Resolution Rules

The *object*_name that is associated with an **overriding** mode shall resolve to statically denote a formal object, or a formal parameter of the associated entity. 4/5

An *object*_name used as a global_name may also be an attribute_reference with attribute designator Storage_Pool. The attribute prefix can denote a formal access-to-object type. 4.1/6

Static Semantics

The presence of the reserved word **overriding** in a global mode indicates that the specification is overriding the mode of a formal parameter with another mode to reflect the overall effect of an invocation of the callable entity on the state associated with the corresponding actual parameter. 5/5

As described in 6.1.2, the following rules are defined in terms of operations that are performed by or on behalf of an entity. 6/5

The Global aspect for a subtype identifies the global variables that can be referenced during default initialization, adjustment as part of assignment, finalization of an object of the subtype, or conversion to the subtype, including the evaluation of any assertion expressions that apply. For a formal access-to-object 7/6

subtype, or an access-to-object subtype whose storage pool is not a standard storage pool, the Global aspect identifies the global variables (other than the storage pool itself) that can be referenced by a call on Allocate, Deallocate, or Storage_Size for the storage pool type. If not specified for the first subtype of a derived type, the aspect defaults to that of the ancestor subtype; if not specified for a nonderived composite first subtype the aspect defaults to that of the enclosing library unit; if not specified for a nonderived elementary first subtype (or scalar base subtype), the aspect defaults to **null** in the absence of a predicate (or when the predicate is statically True), and to that of the enclosing library unit otherwise. If not specified for a nonfirst subtype *S*, the Global aspect defaults to that of the subtype identified in the subtype_indication defining *S*.

8/5 The Global'Class aspect may be specified for the first subtype of a tagged type *T*, indicating an upper bound on the Global aspect of any descendant of *T*. If not specified, it defaults to Unspecified.

Legality Rules

9/5 For a tagged subtype *T*, each mode of its Global aspect shall identify a subset of the variables identified either by the corresponding mode, or by the **in out** mode, of the Global'Class aspect of the first subtype of any ancestor of *T*.

H.7.1 The Use_Formal and Dispatching Aspects

1/5 The Use_Formal and Dispatching aspects are provided to more precisely describe the use of generic formal parameters and dispatching calls within the execution of an operation, enabling more precise checking of conformance with the Nonblocking and global aspects that apply at the point of invocation of the operation.

2/5 For any declaration within a generic unit for which a global or Nonblocking aspect may be specified, other than a generic_formal_parameter_declaration, the following aspect may be specified to indicate which generic formal parameters are *used* by the associated entity:

3/5 Use_Formal The aspect is specified with a formal_parameter_set, with the following form:

4/5
```
formal_parameter_set ::=
   formal_group_designator
 | formal_parameter_name
 | (formal_parameter_name{, formal_parameter_name})
```

5/5
```
formal_group_designator ::= null | all
```

6/5
```
formal_parameter_name ::=
   formal_subtype_mark
 | formal_subprogram_name
 | formal_access_to_subprogram_object_name
```

7/5 For any declaration for which a global or Nonblocking aspect may be specified, other than for a library package, a generic library package, or a generic formal, the following aspect may be specified:

8/5 Dispatching The aspect is specified with a dispatching_operation_set, with the following form:

9/5
```
dispatching_operation_set ::=
   dispatching_operation_specifier
 | (dispatching_operation_specifier{, dispatching_operation_specifier})
```

10/5
```
dispatching_operation_specifier ::=
   dispatching_operation_name (object_name)
```

Name Resolution Rules

A formal_parameter_name in a Use_Formal aspect shall resolve to statically denote a formal subtype, a formal subprogram, or a formal object of an anonymous access-to-subprogram type of an enclosing generic unit or visible formal package. 11/5

The *object_*name of a dispatching_operation_specifier shall resolve to statically name an object (including possibly a formal parameter) of a tagged class-wide type *T*'Class, or of an access type designating a tagged class-wide type *T*'Class; the *dispatching_operation_*name of the dispatching_operation_specifier shall resolve to statically denote a dispatching operation associated with *T*. 12/5

Static Semantics

The *formal parameter set* is identified by a set of formal_parameter_names. Alternatively, the reserved word **null** may be used to indicate none of the generic formal parameters, or **all** to indicate all of the generic formal parameters, of any enclosing generic unit (or visible formal package) can be used within the execution of the operation. If there is no formal parameter set specified for an entity declared within a generic unit, it defaults to **all**. 13/5

The *dispatching operation set* is identified by a set of dispatching_operation_specifiers. It indicates that the Nonblocking and global effects of dispatching calls that match one of the specifiers, rather than being accounted for by the Nonblocking or global aspect, are instead to be accounted for by the invoker of the operation. A dispatching call matches a dispatching_operation_specifier if the name or prefix of the call statically denotes the same operation(s) as that of the dispatching_operation_specifier, and at least one of the objects controlling the call is denoted by, or designated by, a name that statically names the same object as that denoted by the *object_*name of the dispatching_operation_specifier. 14/5

In the absence of any dispatching_operation_specifiers, or if none of them match a dispatching call *C* within an operation *P*, Nonblocking and global aspects checks are performed at the point of the call *C* within *P* using the Nonblocking and Global'Class aspects that apply to the dispatching operation named in call *C*. If there is a match, any global access or potential blocking within the subprogram body invoked by the call *C* is ignored at the point of call within *P*. Instead, when the operation *P* itself is invoked, Nonblocking and global aspect checks are performed presuming each named dispatching operation is called at least once (with the named object controlling the call), but similarly ignoring those dispatching calls that would match a dispatching_operation_specifier applicable at the point of invocation of *P*. 15/5

Legality Rules

Within an operation to which a Use_Formal aspect applies, if the formal parameter set is anything but **all**, then the only generic formal subtypes that may be used, the only formal subprograms that may be called, and the only formal objects of an anonymous access-to-subprogram type that may be dereferenced as part of a call or passed as the actual for an access parameter, are those included in the formal parameter set. 16/5

When an operation (or instance thereof) to which a Use_Formal aspect applies is invoked, Nonblocking and global aspect checks are performed presuming each generic formal parameter (or corresponding actual parameter) of the formal parameter set is used at least once. 17/5

Examples

An example of use of the Dispatching aspect: 18/5

19/5
```
   procedure My_Write(  --  see 13.13.2
      Stream : not null access Ada.Streams.Root_Stream_Type'Class;
      Item   : My_Integer'Base)
      with Dispatching => Write(Stream);
   for My_Integer'Write use My_Write;
```

20/5 *For examples of use of the Use_Formal aspect, see the Element functions of Hashed_Sets in A.18.8.*

Annex J
(normative)
Obsolescent Features

This Annex contains descriptions of features of the language whose functionality is largely redundant with other features defined by this Reference Manual. Use of these features is not recommended in newly written programs. Use of these features can be prevented by using pragma Restrictions (No_Obsolescent_Features), see 13.12.1. 1/2

J.1 Renamings of Library Units

Static Semantics

The following library_unit_renaming_declarations exist: 1

```
with Ada.Unchecked_Conversion;
generic function Unchecked_Conversion renames Ada.Unchecked_Conversion;
```
2

```
with Ada.Unchecked_Deallocation;
generic procedure Unchecked_Deallocation renames Ada.Unchecked_Deallocation;
```
3

```
with Ada.Sequential_IO;
generic package Sequential_IO renames Ada.Sequential_IO;
```
4

```
with Ada.Direct_IO;
generic package Direct_IO renames Ada.Direct_IO;
```
5

```
with Ada.Text_IO;
package Text_IO renames Ada.Text_IO;
```
6

```
with Ada.IO_Exceptions;
package IO_Exceptions renames Ada.IO_Exceptions;
```
7

```
with Ada.Calendar;
package Calendar renames Ada.Calendar;
```
8

```
with System.Machine_Code;
package Machine_Code renames System.Machine_Code; -- If supported.
```
9

Implementation Requirements

The implementation shall allow the user to replace these renamings. 10

J.2 Allowed Replacements of Characters

Syntax

The following replacements are allowed for the vertical line, number sign, and quotation mark characters: 1

- A vertical line character (|) can be replaced by an exclamation mark (!) where used as a delimiter. 2
- The number sign characters (#) of a based_literal can be replaced by colons (:) provided that the replacement is done for both occurrences. 3
- The quotation marks (") used as string brackets at both ends of a string literal can be replaced by percent signs (%) provided that the enclosed sequence of characters contains no quotation mark, and provided that both string brackets are replaced. Any percent sign 4

within the sequence of characters shall then be doubled and each such doubled percent sign is interpreted as a single percent sign character value.

5 These replacements do not change the meaning of the program.

J.3 Reduced Accuracy Subtypes

1 A digits_constraint may be used to define a floating point subtype with a new value for its requested decimal precision, as reflected by its Digits attribute. Similarly, a delta_constraint may be used to define an ordinary fixed point subtype with a new value for its *delta*, as reflected by its Delta attribute.

Syntax

2/4 delta_constraint ::= **delta** *static_*simple_expression [range_constraint]

Name Resolution Rules

3/4 The simple_expression of a delta_constraint is expected to be of any real type.

Legality Rules

4/4 The simple_expression of a delta_constraint shall be static.

5 For a subtype_indication with a delta_constraint, the subtype_mark shall denote an ordinary fixed point subtype.

6 For a subtype_indication with a digits_constraint, the subtype_mark shall denote either a decimal fixed point subtype or a floating point subtype (notwithstanding the rule given in 3.5.9 that only allows a decimal fixed point subtype).

Static Semantics

7/4 A subtype_indication with a subtype_mark that denotes an ordinary fixed point subtype and a delta_constraint defines an ordinary fixed point subtype with a *delta* given by the value of the simple_expression of the delta_constraint. If the delta_constraint includes a range_constraint, then the ordinary fixed point subtype is constrained by the range_constraint.

8/4 A subtype_indication with a subtype_mark that denotes a floating point subtype and a digits_constraint defines a floating point subtype with a requested decimal precision (as reflected by its Digits attribute) given by the value of the simple_expression of the digits_constraint. If the digits_constraint includes a range_constraint, then the floating point subtype is constrained by the range_constraint.

Dynamic Semantics

9/4 A delta_constraint is *compatible* with an ordinary fixed point subtype if the value of the simple_expression is no less than the *delta* of the subtype, and the range_constraint, if any, is compatible with the subtype.

10/4 A digits_constraint is *compatible* with a floating point subtype if the value of the simple_expression is no greater than the requested decimal precision of the subtype, and the range_constraint, if any, is compatible with the subtype.

11 The elaboration of a delta_constraint consists of the elaboration of the range_constraint, if any.

J.4 The Constrained Attribute

Static Semantics

For every private subtype S, the following attribute is defined: 1

S'Constrained 2

Yields the value False if S denotes an unconstrained nonformal private subtype with discriminants; also yields the value False if S denotes a generic formal private subtype, and the associated actual subtype is either an unconstrained subtype with discriminants or an unconstrained array subtype; yields the value True otherwise. The value of this attribute is of the predefined subtype Boolean.

J.5 ASCII

Static Semantics

The following declaration exists in the declaration of package Standard: 1

```
package ASCII is                                                                    2
   -- Control characters:                                                           3
   NUL    : constant Character := nul;    SOH    : constant Character := soh;     4
   STX    : constant Character := stx;    ETX    : constant Character := etx;
   EOT    : constant Character := eot;    ENQ    : constant Character := enq;
   ACK    : constant Character := ack;    BEL    : constant Character := bel;
   BS     : constant Character := bs;     HT     : constant Character := ht;
   LF     : constant Character := lf;     VT     : constant Character := vt;
   FF     : constant Character := ff;     CR     : constant Character := cr;
   SO     : constant Character := so;     SI     : constant Character := si;
   DLE    : constant Character := dle;    DC1    : constant Character := dc1;
   DC2    : constant Character := dc2;    DC3    : constant Character := dc3;
   DC4    : constant Character := dc4;    NAK    : constant Character := nak;
   SYN    : constant Character := syn;    ETB    : constant Character := etb;
   CAN    : constant Character := can;    EM     : constant Character := em;
   SUB    : constant Character := sub;    ESC    : constant Character := esc;
   FS     : constant Character := fs;     GS     : constant Character := gs;
   RS     : constant Character := rs;     US     : constant Character := us;
   DEL    : constant Character := del;
   -- Other characters:                                                          5
   Exclam   : constant Character:= '!'; Quotation : constant Character:= '"';    6
   Sharp    : constant Character:= '#'; Dollar    : constant Character:= '$';
   Percent  : constant Character:= '%'; Ampersand : constant Character:= '&';
   Colon    : constant Character:= ':'; Semicolon : constant Character:= ';';
   Query    : constant Character:= '?'; At_Sign   : constant Character:= '@';
   L_Bracket: constant Character:= '['; Back_Slash: constant Character:= '\';
   R_Bracket: constant Character:= ']'; Circumflex: constant Character:= '^';
   Underline: constant Character:= '_'; Grave     : constant Character:= '`';
   L_Brace  : constant Character:= '{'; Bar       : constant Character:= '|';
   R_Brace  : constant Character:= '}'; Tilde     : constant Character:= '~';
   -- Lower case letters:                                                        7
   LC_A: constant Character:= 'a';                                              8
   ...
   LC_Z: constant Character:= 'z';
end ASCII;                                                                      9
```

J.6 Numeric_Error

Static Semantics

1 The following declaration exists in the declaration of package Standard:

2
```
Numeric_Error : exception renames Constraint_Error;
```

J.7 At Clauses

Syntax

1 at_clause ::= **for** direct_name **use at** expression;

Static Semantics

2 An at_clause of the form "for *x* use at *y*;" is equivalent to an attribute_definition_clause of the form "for *x*'Address use *y*;".

J.7.1 Interrupt Entries

1 Implementations are permitted to allow the attachment of task entries to interrupts via the address clause. Such an entry is referred to as an *interrupt entry*.

2 The address of the task entry corresponds to a hardware interrupt in an implementation-defined manner. (See Ada.Interrupts.Reference in C.3.2.)

Static Semantics

3 The following attribute is defined:

4 For any task entry X:

5 X'Address — For a task entry whose address is specified (an *interrupt entry*), the value refers to the corresponding hardware interrupt. For such an entry, as for any other task entry, the meaning of this value is implementation defined. The value of this attribute is of the type of the subtype System.Address.

6 Address may be specified for single entries via an attribute_definition_clause.

Dynamic Semantics

7 As part of the initialization of a task object, the address clause for an interrupt entry is elaborated, which evaluates the expression of the address clause. A check is made that the address specified is associated with some interrupt to which a task entry may be attached. If this check fails, Program_Error is raised. Otherwise, the interrupt entry is attached to the interrupt associated with the specified address.

8 Upon finalization of the task object, the interrupt entry, if any, is detached from the corresponding interrupt and the default treatment is restored.

9 While an interrupt entry is attached to an interrupt, the interrupt is reserved (see C.3).

10 An interrupt delivered to a task entry acts as a call to the entry issued by a hardware task whose priority is in the System.Interrupt_Priority range. It is implementation defined whether the call is performed as an ordinary entry call, a timed entry call, or a conditional entry call; which kind of call is performed can depend on the specific interrupt.

Bounded (Run-Time) Errors

It is a bounded error to evaluate E'Caller (see C.7.1) in an accept_statement for an interrupt entry. The possible effects are the same as for calling Current_Task from an entry body. 11

Documentation Requirements

The implementation shall document to which interrupts a task entry may be attached. 12

The implementation shall document whether the invocation of an interrupt entry has the effect of an ordinary entry call, conditional call, or a timed call, and whether the effect varies in the presence of pending interrupts. 13

Implementation Permissions

The support for this subclause is optional. 14

Interrupts to which the implementation allows a task entry to be attached may be designated as reserved for the entire duration of program execution; that is, not just when they have an interrupt entry attached to them. 15

Interrupt entry calls may be implemented by having the hardware execute directly the appropriate accept_statement. Alternatively, the implementation is allowed to provide an internal interrupt handler to simulate the effect of a normal task calling the entry. 16/1

The implementation is allowed to impose restrictions on the specifications and bodies of tasks that have interrupt entries. 17

It is implementation defined whether direct calls (from the program) to interrupt entries are allowed. 18

If a select_statement contains both a terminate_alternative and an accept_alternative for an interrupt entry, then an implementation is allowed to impose further requirements for the selection of the terminate_alternative in addition to those given in 9.3. 19

NOTE 1 Queued interrupts correspond to ordinary entry calls. Interrupts that are lost if not immediately processed correspond to conditional entry calls. It is a consequence of the priority rules that an accept_statement executed in response to an interrupt can be executed with the active priority at which the hardware generates the interrupt, taking precedence over lower priority tasks, without a scheduling action. 20/1

NOTE 2 Control information that is supplied upon an interrupt can be passed to an associated interrupt entry as one or more parameters of mode **in**. 21

Examples

Example of an interrupt entry: 22

```
task Interrupt_Handler is                                                    23
  entry Done;
  for Done'Address use
Ada.Interrupts.Reference(Ada.Interrupts.Names.Device_Done);
end Interrupt_Handler;
```

J.8 Mod Clauses

Syntax

mod_clause ::= **at mod** *static_*expression; 1

Static Semantics

2 A record_representation_clause of the form:

3/3
```
for r use
    record at mod a;
        ...
    end record;
```

4 is equivalent to:

5
```
for r'Alignment use a;
for r use
    record
        ...
    end record;
```

J.9 The Storage_Size Attribute

Static Semantics

1 For any task subtype T, the following attribute is defined:

2 T'Storage_Size

Denotes an implementation-defined value of type *universal_integer* representing the number of storage elements reserved for a task of the subtype T.

3/3 Storage_Size may be specified for a task first subtype that is not an interface via an attribute_definition_clause. When the attribute is specified, the Storage_Size aspect is specified to be the value of the given expression.

J.10 Specific Suppression of Checks

1/2 Pragma Suppress can be used to suppress checks on specific entities.

Syntax

2/2 The form of a specific Suppress pragma is as follows:

3/2 **pragma** Suppress(identifier, [On =>] name);

Legality Rules

4/2 The identifier shall be the name of a check (see 11.5). The name shall statically denote some entity.

5/2 For a specific Suppress pragma that is immediately within a package_specification, the name shall denote an entity (or several overloaded subprograms) declared immediately within the package_specification.

Static Semantics

6/2 A specific Suppress pragma applies to the named check from the place of the pragma to the end of the innermost enclosing declarative region, or, if the pragma is given in a package_specification, to the end of the scope of the named entity. The pragma applies only to the named entity, or, for a subtype, on objects and values of its type. A specific Suppress pragma suppresses the named check for any entities to which it applies (see 11.5). Which checks are associated with a specific entity is not defined by this document.

Implementation Permissions

7/2 An implementation is allowed to place restrictions on specific Suppress pragmas.

NOTE An implementation can support a similar On parameter on pragma Unsuppress (see 11.5). 8/5

J.11 The Class Attribute of Untagged Incomplete Types

Static Semantics

For the first subtype S of a type *T* declared by an incomplete_type_declaration that is not tagged, the following attribute is defined: 1/2

S'Class Denotes the first subtype of the incomplete class-wide type rooted at *T*. The completion of *T* shall declare a tagged type. Such an attribute reference shall occur in the same library unit as the incomplete_type_declaration. 2/2

J.12 Pragma Interface

Syntax

In addition to an identifier, the reserved word **interface** is allowed as a pragma name, to provide compatibility with a prior edition of this Reference Manual. 1/2

J.13 Dependence Restriction Identifiers

The following restrictions involve dependence on specific language-defined units. The more general restriction No_Dependence (see 13.12.1) should be used for this purpose. 1/2

Static Semantics

The following *restriction*_identifiers exist: 2/2

No_Asynchronous_Control 3/2
Semantic dependence on the predefined package Asynchronous_Task_Control is not allowed.

No_Unchecked_Conversion 4/2
Semantic dependence on the predefined generic function Unchecked_Conversion is not allowed.

No_Unchecked_Deallocation 5/2
Semantic dependence on the predefined generic procedure Unchecked_Deallocation is not allowed.

J.14 Character and Wide_Character Conversion Functions

Static Semantics

The following declarations exist in the declaration of package Ada.Characters.Handling: 1/2

```
function Is_Character (Item : in Wide_Character) return Boolean                    2/2
   renames Conversions.Is_Character;
function Is_String    (Item : in Wide_String)    return Boolean
   renames Conversions.Is_String;

function To_Character (Item       : in Wide_Character;                             3/2
                       Substitute : in Character := ' ')
                       return Character
   renames Conversions.To_Character;
```

4/2
```
function To_String    (Item       : in Wide_String;
                       Substitute : in Character := ' ')
                       return String
   renames Conversions.To_String;
```

5/2
```
function To_Wide_Character (Item : in Character) return Wide_Character
   renames Conversions.To_Wide_Character;
```

6/2
```
function To_Wide_String    (Item : in String)    return Wide_String
   renames Conversions.To_Wide_String;
```

J.15 Aspect-related Pragmas

1/6 Pragmas can be used as an alternative to aspect_specifications to specify certain aspects. Such pragmas are aspect-related pragmas (see 2.8).

Name Resolution Rules

2/5 Certain pragmas are defined to be *program unit pragmas*. A name given as the argument of a program unit pragma shall resolve to denote the declarations or renamings of one or more program units that occur immediately within the declarative region or compilation in which the pragma immediately occurs, or it shall resolve to denote the declaration of the immediately enclosing program unit (if any); the pragma applies to the denoted program unit(s). If there are no names given as arguments, the pragma applies to the immediately enclosing program unit.

Legality Rules

3/5 A program unit pragma shall appear in one of these places:

4/5 - At the place of a compilation_unit, in which case the pragma shall immediately follow in the same compilation (except for other pragmas) a library_unit_declaration that is a subprogram_declaration, generic_subprogram_declaration, or generic_instantiation, and the pragma shall have an argument that is a name denoting that declaration.

5/5 - Immediately within the visible part of a program unit and before any nested declaration (but not within a generic formal part), in which case the argument, if any, shall be a direct_name that denotes the immediately enclosing program unit declaration.

6/5 - At the place of a declaration other than the first, of a declarative_part or program unit declaration, in which case the pragma shall have an argument, which shall be a direct_name that denotes one or more of the following (and nothing else): a subprogram_declaration, a generic_subprogram_declaration, or a generic_instantiation, of the same declarative_part or program unit declaration.

7/5 Certain program unit pragmas are defined to be *library unit pragmas*. If a library unit pragma applies to a program unit, the program unit shall be a library unit.

Static Semantics

8/5 A library unit pragma that applies to a generic unit does not apply to its instances, unless a specific rule for the pragma specifies the contrary.

Implementation Advice

9/5 When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance.

J.15.1 Pragma Inline

Syntax

The form of a pragma Inline, which is a program unit pragma (see 10.1.5), is as follows: 1/3

pragma Inline (name{, name}); 2/3

Legality Rules

The pragma shall apply to one or more callable entities or generic subprograms. 3/3

Static Semantics

Pragma Inline specifies that the Inline aspect (see 6.3.2) for each entity denoted by each name given in the pragma has the value True. 4/3

Implementation Permissions

An implementation may allow a pragma Inline that has an argument which is a direct_name denoting a subprogram_body of the same declarative_part. 5/3

NOTE The name in a pragma Inline can denote more than one entity in the case of overloading. Such a pragma applies to all of the denoted entities. 6/5

J.15.2 Pragma No_Return

Syntax

The form of a pragma No_Return, which is a representation pragma (see 13.1), is as follows: 1/3

pragma No_Return (*subprogram*_local_name{, *subprogram*_local_name}); 2/5

Legality Rules

Each *subprogram*_local_name shall denote one or more subprograms or generic subprograms. The *subprogram*_local_name shall not denote a null procedure nor an instance of a generic unit. 3/5

Static Semantics

Pragma No_Return specifies that the No_Return aspect (see 6.5.1) for each subprogram denoted by each local_name given in the pragma has the value True. 4/5

J.15.3 Pragma Pack

Syntax

The form of a pragma Pack, which is a representation pragma (see 13.1), is as follows: 1/3

pragma Pack (*first_subtype*_local_name); 2/3

Legality Rules

The *first_subtype*_local_name of a pragma Pack shall denote a composite subtype. 3/3

Static Semantics

Pragma Pack specifies that the Pack aspect (see 13.2) for the type denoted by *first_subtype*_local_name has the value True. 4/3

J.15.4 Pragma Storage_Size

Syntax

1/3 The form of a pragma Storage_Size is as follows:

2/3 **pragma** Storage_Size (expression);

3/3 A pragma Storage_Size is allowed only immediately within a task_definition.

Name Resolution Rules

4/3 The expression of a pragma Storage_Size is expected to be of any integer type.

Static Semantics

5/3 The pragma Storage_Size sets the Storage_Size aspect (see 13.3) of the type defined by the immediately enclosing task_definition to the value of the expression of the pragma.

J.15.5 Interfacing Pragmas

Syntax

1/3 An *interfacing pragma* is a representation pragma that is one of the pragmas Import, Export, or Convention. Their forms are as follows:

2/3 **pragma** Import(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]);

3/3 **pragma** Export(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]);

4/3 **pragma** Convention([Convention =>] *convention*_identifier,[Entity =>] local_name);

5/3 For pragmas Import and Export, the argument for Link_Name shall not be given without the *pragma_argument*_identifier unless the argument for External_Name is given.

Name Resolution Rules

6/3 The expected type for an *external_name_string*_expression and a *link_name_string*_expression in an interfacing pragma is String.

Legality Rules

7/3 The *convention*_identifier of an interfacing pragma shall be the name of a convention (see B.1).

8/3 A pragma Import shall be the completion of a declaration. Notwithstanding any rule to the contrary, a pragma Import may serve as the completion of any kind of (explicit) declaration if supported by an implementation for that kind of declaration. If a completion is a pragma Import, then it shall appear in the same declarative_part, package_specification, task_definition, or protected_definition as the declaration. For a library unit, it shall appear in the same compilation, before any subsequent compilation_units other

than pragmas. If the local_name denotes more than one entity, then the pragma Import is the completion of all of them.

The *external_name_string*_expression and *link_name_string*_expression of a pragma Import or Export shall be static. 9/3

The local_name of each of these pragmas shall denote a declaration that may have the similarly named aspect specified. 10/3

Static Semantics

An interfacing pragma specifies various aspects of the entity denoted by the local_name as follows: 11/3

- The Convention aspect (see B.1) is *convention*_identifier. 12/3
- A pragma Import specifies that the Import aspect (see B.1) is True. 13/3
- A pragma Export specifies that the Export aspect (see B.1) is True. 14/3
- For both pragma Import and Export, if an external name is given in the pragma, the External_Name aspect (see B.1) is specified to be *external_name_string*_expression. If a link name is given in the pragma, the Link_Name aspect (see B.1) is specified to be the *link_name_string*_expression. 15/3

J.15.6 Pragma Unchecked_Union

Syntax

The form of a pragma Unchecked_Union, which is a representation pragma (see 13.1), is as follows: 1/3

pragma Unchecked_Union (*first_subtype*_local_name); 2/3

Legality Rules

The *first_subtype*_local_name of a pragma Unchecked_Union shall denote an unconstrained discriminated record subtype having a variant_part. 3/3

Static Semantics

A pragma Unchecked_Union specifies that the Unchecked_Union aspect (see B.3.3) for the type denoted by *first_subtype*_local_name has the value True. 4/3

J.15.7 Pragmas Interrupt_Handler and Attach_Handler

Syntax

The form of a pragma Interrupt_Handler is as follows: 1/3

pragma Interrupt_Handler (*handler*_name); 2/3

The form of a pragma Attach_Handler is as follows: 3/3

pragma Attach_Handler (*handler*_name, expression); 4/3

Name Resolution Rules

For the Interrupt_Handler and Attach_Handler pragmas, the *handler*_name shall resolve to denote a protected procedure with a parameterless profile. 5/3

For the Attach_Handler pragma, the expected type for the expression is Interrupts.Interrupt_Id (see C.3.2). 6/3

Legality Rules

7/3 The Attach_Handler and Interrupt_Handler pragmas are only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library-level declaration, and shall not be declared within a generic body. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

Static Semantics

8/3 For an implementation that supports Annex C, a pragma Interrupt_Handler specifies the Interrupt_Handler aspect (see C.3.1) for the protected procedure *handler*_name to have the value True. For an implementation that supports Annex C, a pragma Attach_Handler specifies the Attach_Handler aspect (see C.3.1) for the protected procedure *handler*_name to have the value of the given expression as evaluated at object creation time.

J.15.8 Shared Variable Pragmas

Syntax

1/5 The following pragmas are defined with the given forms:

2/3 **pragma** Atomic (local_name);

3/3 **pragma** Volatile (local_name);

4/3 **pragma** Independent (*component*_local_name);

5/3 **pragma** Atomic_Components (*array*_local_name);

6/3 **pragma** Volatile_Components (*array*_local_name);

7/3 **pragma** Independent_Components (local_name);

Name Resolution Rules

8/3 The local_name in an Atomic or Volatile pragma shall resolve to denote either an object_declaration, a noninherited component_declaration, or a full_type_declaration. The *component*_local_name in an Independent pragma shall resolve to denote a noninherited component_declaration. The *array*_local_name in an Atomic_Components or Volatile_Components pragma shall resolve to denote the declaration of an array type or an array object of an anonymous type. The local_name in an Independent_Components pragma shall resolve to denote the declaration of an array or record type or an array object of an anonymous type.

Static Semantics

9/3 These pragmas are representation pragmas (see 13.1). Each of these pragmas specifies that the similarly named aspect (see C.6) of the type, object, or component denoted by its argument is True.

Legality Rules

10/3 The local_name of each of these pragmas shall denote a declaration that may have the similarly named aspect specified.

J.15.9 Pragma CPU

Syntax

The form of a pragma CPU is as follows: 1/3

pragma CPU (expression); 2/3

Name Resolution Rules

The expected type for the expression of a pragma CPU is System.Multiprocessors.CPU_Range. 3/3

Legality Rules

A CPU pragma is allowed only immediately within a task_definition, protected_definition, or the declarative_part of a subprogram_body. 4/5

For a CPU pragma that appears in the declarative_part of a subprogram_body, the expression shall be static. 5/3

Static Semantics

For an implementation that supports Annex D, a pragma CPU specifies the value of the CPU aspect (see D.16). If the pragma appears in a task_definition, the expression is associated with the aspect for the task type or single_task_declaration that contains the pragma. If the pragma appears in a protected_definition, the expression is associated with the aspect for the protected type or single_protected_declaration that contains the pragma. Otherwise, the expression is associated with the aspect for the subprogram that contains the pragma. 6/5

J.15.10 Pragma Dispatching_Domain

Syntax

The form of a pragma Dispatching_Domain is as follows: 1/3

pragma Dispatching_Domain (expression); 2/3

Name Resolution Rules

The expected type for the expression is System.Multiprocessors.Dispatching_Domains.-Dispatching_Domain. 3/3

Legality Rules

A Dispatching_Domain pragma is allowed only immediately within a task_definition. 4/3

Static Semantics

For an implementation that supports Annex D, a pragma Dispatching_Domain specifies the value of the Dispatching_Domain aspect (see D.16.1). The expression is associated with the aspect for the task type or single_task_declaration that contains the pragma. 5/3

J.15.11 Pragmas Priority and Interrupt_Priority

Syntax

1/3 The form of a pragma Priority is as follows:

2/3 **pragma** Priority (expression);

3/3 The form of a pragma Interrupt_Priority is as follows:

4/3 **pragma** Interrupt_Priority [(expression);]

Name Resolution Rules

5/3 The expected type for the expression in a Priority or Interrupt_Priority pragma is Integer.

Legality Rules

6/3 A Priority pragma is allowed only immediately within a task_definition, a protected_definition, or the declarative_part of a subprogram_body. An Interrupt_Priority pragma is allowed only immediately within a task_definition or a protected_definition.

7/3 For a Priority pragma that appears in the declarative_part of a subprogram_body, the expression shall be static, and its value shall be in the range of System.Priority.

Static Semantics

8/3 For an implementation that supports Annex D, a pragma Priority specifies the value of the Priority aspect (see D.1) and a pragma Interrupt_Priority specifies the value of the Interrupt_Priority aspect as follows:

9/3 - If the pragma appears in a task_definition, the expression is associated with the aspect for the task type or single_task_declaration that contains the pragma;

10/3 - If the pragma appears in a protected_definition, the expression is associated with the aspect for the protected type or single_protected_declaration that contains the pragma;

11/3 - If the pragma appears in the declarative_part of a subprogram_body, the expression is associated with the aspect for the subprogram that contains the pragma.

12/3 If there is no expression in an Interrupt_Priority pragma, the Interrupt_Priority aspect has the value Interrupt_Priority'Last.

J.15.12 Pragma Relative_Deadline

Syntax

1/3 The form of a pragma Relative_Deadline is as follows:

2/3 **pragma** Relative_Deadline (*relative_deadline*_expression);

Name Resolution Rules

3/3 The expected type for a *relative_deadline*_expression is Real_Time.Time_Span.

Legality Rules

4/3 A Relative_Deadline pragma is allowed only immediately within a task_definition or the declarative_part of a subprogram_body.

Static Semantics

For an implementation that supports Annex D, a pragma Relative_Deadline specifies the value of the Relative_Deadline aspect (see D.2.6). If the pragma appears in a task_definition, the expression is associated with the aspect for the task type or single_task_declaration that contains the pragma; otherwise, the expression is associated with the aspect for the subprogram that contains the pragma. 5/3

J.15.13 Pragma Asynchronous

Syntax

The form of a pragma Asynchronous, which is a representation pragma (see 13.1), is as follows: 1/3

pragma Asynchronous (local_name); 2/3

Static Semantics

For an implementation that supports Annex E, a pragma Asynchronous specifies that the Asynchronous aspect (see E.4.1) for the procedure or type denoted by local_name has the value True. 3/3

Legality Rules

The local_name of a pragma Asynchronous shall denote a declaration that may have aspect Asynchronous specified. 4/3

J.15.14 Elaboration Control Pragmas

This subclause defines pragmas that specify aspects that help control the elaboration order of library_items. 1/5

Syntax

The following pragmas are defined with the given forms: 2/5

pragma Preelaborate[(*library_unit*_name)]; 3/5

pragma Preelaborable_Initialization(direct_name); 4/5

pragma Pure[(*library_unit*_name)]; 5/5

pragma Elaborate_Body[(*library_unit*_name)]; 6/5

Pragmas Preelaborate, Pure, and Elaborate_Body are library unit pragmas. 7/5

Static Semantics

A pragma Preelaborate specifies that a library unit is preelaborated, namely that the Preelaborate aspect (see 10.2.1) of the library unit is True. 8/5

A pragma Pure specifies that a library unit is declared pure, namely that the Pure aspect (see 10.2.1) of the library unit is True. 9/5

A pragma Elaborate_Body specifies that a library unit requires a completion, namely that the Elaborate_Body aspect (see 10.2.1) of the library unit is True. 10/5

Legality Rules

A pragma Preelaborable_Initialization specifies that the Preelaborable_Initialization aspect (see 10.2.1) for a composite type is True. This pragma shall appear in the visible part of a package or generic package. 11/5

12/5 If the pragma appears in the first declaration list of a package_specification, then the direct_name shall denote the first subtype of a composite type, and the type shall be declared immediately within the same package as the pragma. The composite type shall be one for which the Preelaborable_Initialization aspect can be directly specified as True. In addition to the places where Legality Rules normally apply (see 12.3), these rules also apply in the private part of an instance of a generic unit.

13/5 If the pragma appears in a generic_formal_part, then the direct_name shall denote a type declared in the same generic_formal_part as the pragma, and be one for which the Preelaborable_Initialization aspect can be directly specified as True.

14/5 NOTE Pragmas Elaborate and Elaborate_All, which do not have associated aspects, are found in 10.2.1.

J.15.15 Distribution Pragmas

1/5 This subclause defines pragmas that specify properties of units for distributed systems.

Syntax

2/5 The following pragmas are defined with the given forms:

3/5 **pragma** Shared_Passive[(*library_unit*_name)];

4/5 **pragma** Remote_Types[(*library_unit*_name)];

5/5 **pragma** Remote_Call_Interface[(*library_unit*_name)];

6/5 **pragma** All_Calls_Remote[(*library_unit*_name)];

7/5 Each of these pragmas is a library unit pragma.

Static Semantics

8/5 A *categorization pragma* is a pragma that specifies a corresponding categorization aspect.

9/5 The pragmas Shared_Passive, Remote_Types, and Remote_Call_Interface are categorization pragmas. In addition, the pragma Pure (see J.15.14) is considered a categorization pragma.

10/5 A pragma Shared_Passive specifies that a library unit is a shared passive library unit, namely that the Shared_Passive aspect (see E.2.1) of the library unit is True.

11/5 A pragma Remote_Types specifies that a library unit is a remote types library unit, namely that the Remote_Types aspect (see E.2.2) of the library unit is True.

12 A pragma Remote_Call_Interface specifies that a library unit is a remote call interface, namely that the Remote_Call_Interface aspect (see E.2.3) of the library unit is True.

13 A pragma All_Calls_Remote specifies that the All_Calls_Remote aspect (see E.2.3) of the library unit is True.

Annex K
(informative)
Language-Defined Aspects and Attributes

This annex summarizes the definitions given elsewhere of the language-defined aspects and attributes. Some aspects have corresponding attributes, as noted. 1/3

K.1 Language-Defined Aspects

This subclause summarizes the definitions given elsewhere of the language-defined aspects. Aspects are properties of entities that can be specified by the Ada program; unless otherwise specified below, aspects can be specified using an aspect_specification. 1/3

Address — Machine address of an entity. See 13.3. 2/3

Aggregate — Mechanism to define user-defined aggregates. See 4.3.5. 2.1/5

Alignment (object) 3/3
: Alignment of an object. See 13.3.

Alignment (subtype) 4/3
: Alignment of a subtype. See 13.3.

All_Calls_Remote 5/4
: All indirect or dispatching remote subprogram calls, and all direct remote subprogram calls, should use the Partition Communication Subsystem. See E.2.3.

Allows_Exit — An indication of whether a subprogram will operate correctly for arbitrary transfers of control. See 5.5.3. 5.1/5

Asynchronous — Remote procedure calls are asynchronous; the caller continues without waiting for the call to return. See E.4.1. 6/3

Atomic — Declare that a type, object, or component is atomic. See C.6. 7/3

Atomic_Components 8/3
: Declare that the components of an array type or object are atomic. See C.6.

Attach_Handler 9/3
: Protected procedure is attached to an interrupt. See C.3.1.

Bit_Order — Order of bit numbering in a record_representation_clause. See 13.5.3. 10/3

Coding — Internal representation of enumeration literals. Specified by an enumeration_representation_clause, not by an aspect_specification. See 13.4. 11/3

Component_Size 12/3
: Size in bits of a component of an array type. See 13.3.

Constant_Indexing 13/3
: Defines function(s) to implement user-defined indexed_components. See 4.1.6.

Convention — Calling convention or other convention used for interfacing to other languages. See B.1. 14/3

CPU — Processor on which a given task, or calling task for a protected operation, should run. See D.16. 15/5

Default_Component_Value 16/3
: Default value for the components of an array-of-scalar subtype. See 3.6.

16.1/5 Default_Initial_Condition
A condition that will hold true after the default initialization of an object. See 7.3.3.

17/3 Default_Iterator
Default iterator to be used in **for** loops. See 5.5.1.

18/3 Default_Storage_Pool
Default storage pool for a generic instance. See 13.11.3.

19/3 Default_Value
Default value for a scalar subtype. See 3.5.

19.1/4 Discard_Names
Requests a reduction in storage for names associated with an entity. See C.5.

19.2/5 Dispatching Generic formal parameters used in the implementation of an entity. See H.7.1.

20/3 Dispatching_Domain
Domain (group of processors) on which a given task should run. See D.16.1.

21/5 Dynamic_Predicate
Condition that will hold true for objects of a given subtype; the subtype is not static. See 3.2.4.

22/5 Elaborate_Body
A given package will have a body, and that body is elaborated immediately after the declaration. See 10.2.1.

22.1/4 Exclusive_Functions
Specifies mutual exclusion behavior of protected functions in a protected type. See 9.5.1.

23/3 Export Entity is exported to another language. See B.1.

24/3 External_Name
Name used to identify an imported or exported entity. See B.1.

25/3 External_Tag Unique identifier for a tagged type in streams. See 13.3.

25.1/5 Full_Access_Only
Declare that a volatile type, object, or component is full access. See C.6.

25.2/5 Global Global object usage contract. See 6.1.2.

25.3/5 Global'Class Global object usage contract inherited on derivation. See 6.1.2.

26/3 Implicit_Dereference
Mechanism for user-defined implicit .**all**. See 4.1.5.

27/3 Import Entity is imported from another language. See B.1.

28/3 Independent Declare that a type, object, or component is independently addressable. See C.6.

29/3 Independent_Components
Declare that the components of an array or record type, or an array object, are independently addressable. See C.6.

30/3 Inline For efficiency, Inline calls are requested for a subprogram. See 6.3.2.

31/3 Input Function to read a value from a stream for a given type, including any bounds and discriminants. See 13.13.2.

31.1/4 Input'Class Function to read a value from a stream for a the class-wide type associated with a given type, including any bounds and discriminants. See 13.13.2.

Integer_Literal 31.2/5
Defines a function to implement user-defined integer literals. See 4.2.1.

Interrupt_Handler 32/3
Protected procedure may be attached to interrupts. See C.3.1.

Interrupt_Priority 33/3
Priority of a task object or type, or priority of a protected object or type; the priority is in the interrupt range. See D.1.

Iterator_Element 34/3
Element type to be used for user-defined iterators. See 5.5.1.

Iterator_View 34.1/5
An alternative type to be used for container element iterators. See 5.5.1.

Layout (record) 35/3
Layout of record components. Specified by a record_representation_clause, not by an aspect_specification. See 13.5.1.

Link_Name Linker symbol used to identify an imported or exported entity. See B.1. 36/3

Machine_Radix 37/3
Radix (2 or 10) that is used to represent a decimal fixed point type. See F.1.

Max_Entry_Queue_Length 37.1/5
The maximum entry queue length for a task type, protected type, or entry. See D.4.

No_Controlled_Parts 37.2/5
A specification that a type and its descendants do not have controlled parts. See H.4.1.

No_Return A subprogram will not return normally. See 6.5.1. 38/5

Nonblocking Specifies that an associated subprogram does not block. See 9.5. 38.1/5

Output Procedure to write a value to a stream for a given type, including any bounds and discriminants. See 13.13.2. 39/3

Output'Class Procedure to write a value to a stream for a the class-wide type associated with a given type, including any bounds and discriminants. See 13.13.2. 39.1/4

Pack Minimize storage when laying out records and arrays. See 13.2. 40/3

Parallel_Calls 40.1/5
Specifies whether a given subprogram is expected to be called in parallel. See 9.10.1.

Parallel_Iterator 40.2/5
An indication of whether a subprogram may use multiple threads of control to invoke a loop body procedure. See 5.5.3.

Post Postcondition; a condition that will hold true after a call. See 6.1.1. 41/5

Post'Class Postcondition that applies to corresponding subprograms of descendant types. See 6.1.1. 42/5

Pre Precondition; a condition that is expected to hold true before a call. See 6.1.1. 43/5

Pre'Class Precondition that applies to corresponding subprograms of descendant types. See 6.1.1. 44/5

Predicate_Failure 44.1/4
Action to be performed when a predicate check fails. See 3.2.4.

Preelaborable_Initialization 44.2/5
Declares that a type has preelaborable initialization. See 10.2.1.

Preelaborate Code execution during elaboration is avoided for a given package. See 10.2.1. 45/3

46/3 Priority — Priority of a task object or type, or priority of a protected object or type; the priority is not in the interrupt range. See D.1.

47/3 Pure — Side effects are avoided in the subprograms of a given package. See 10.2.1.

47.1/5 Put_Image — Procedure to define the image of a given type. See 4.10.

48/3 Read — Procedure to read a value from a stream for a given type. See 13.13.2.

48.1/4 Read'Class — Procedure to read a value from a stream for the class-wide type associated with a given type. See 13.13.2.

48.2/5 Real_Literal — Defines a function or functions to implement user-defined real literals. See 4.2.1.

49/3 Record layout — See Layout. See 13.5.1.

50/5 Relative_Deadline
Task or protected type parameter used in Earliest Deadline First Dispatching. See D.2.6.

51/3 Remote_Call_Interface
Subprograms in a given package may be used in remote procedure calls. See E.2.3.

52/3 Remote_Types
Types in a given package may be used in remote procedure calls. See E.2.2.

53/3 Shared_Passive
A given package is used to represent shared memory in a distributed system. See E.2.1.

54/3 Size (object) — Size in bits of an object. See 13.3.

55/3 Size (subtype)
Size in bits of a subtype. See 13.3.

56/3 Small — Scale factor for a fixed point type. See 3.5.10.

56.1/5 Stable_Properties
A list of functions describing characteristics that usually are unchanged by primitive operations of the type or an individual primitive subprogram. See 7.3.4.

56.2/5 Stable_Properties'Class
A list of functions describing characteristics that usually are unchanged by primitive operations of a class of types or a primitive subprogram for such a class. See 7.3.4.

56.3/5 Static — Specifies that an associated expression function can be used in static expressions. See 6.8.

57/5 Static_Predicate
Condition that will hold true for objects of a given subtype; the subtype may be static. See 3.2.4.

58/3 Storage_Pool — Pool of memory from which **new** will allocate for a given access type. See 13.11.

59/3 Storage_Size (access)
Sets memory size for allocations for an access type. See 13.11.

60/3 Storage_Size (task)
Size in storage elements reserved for a task type or single task object. See 13.3.

61/3 Stream_Size — Size in bits used to represent elementary objects in a stream. See 13.13.2.

61.1/5 String_Literal
Defines a function to implement user-defined string literals. See 4.2.1.

62/5 Synchronization
Defines whether a given primitive operation of a synchronized interface will be implemented by an entry or protected procedure. See 9.5.

Type_Invariant 63/3
A condition that will hold true for all objects of a type. See 7.3.2.

Type_Invariant'Class 64/3
A condition that will hold true for all objects in a class of types. See 7.3.2.

Unchecked_Union 65/3
Type is used to interface to a C union type. See B.3.3.

Use_Formal Generic formal parameters used in the implementation of an entity. See H.7.1. 65.1/5

Variable_Indexing 66/3
Defines function(s) to implement user-defined indexed_components. See 4.1.6.

Volatile Declare that a type, object, or component is volatile. See C.6. 67/3

Volatile_Components 68/3
Declare that the components of an array type or object are volatile. See C.6.

Write Procedure to write a value to a stream for a given type. See 13.13.2. 69/3

Write'Class Procedure to write a value to a stream for a the class-wide type associated with a given type. See 13.13.2. 69.1/4

Yield Ensures that a callable entity includes a task dispatching point. See D.2.1. 69.2/5

K.2 Language-Defined Attributes

This subclause summarizes the definitions given elsewhere of the language-defined attributes. Attributes are properties of entities that can be queried by an Ada program. 1/3

P'Access For a prefix P that denotes a subprogram: 2

P'Access yields an access value that designates the subprogram denoted by P. The type of P'Access is an access-to-subprogram type (*S*), as determined by the expected type. See 3.10.2. 3

X'Access For a prefix X that denotes an aliased view of an object: 4

X'Access yields an access value that designates the object denoted by X. The type of X'Access is an access-to-object type, as determined by the expected type. The expected type shall be a general access type. See 3.10.2. 5

X'Address For a prefix X that denotes an object, program unit, or label: 6/1

Denotes the address of the first of the storage elements allocated to X. For a program unit or label, this value refers to the machine code associated with the corresponding body or statement. The value of this attribute is of type System.Address. See 13.3. 7

S'Adjacent For every subtype S of a floating point type *T*: 8

S'Adjacent denotes a function with the following specification: 9

```
function S'Adjacent (X, Towards : T)
  return T
```
10

If *Towards* = *X*, the function yields *X*; otherwise, it yields the machine number of the type *T* adjacent to *X* in the direction of *Towards*, if that machine number exists. If the result would be outside the base range of S, Constraint_Error is raised. When *T*'Signed_Zeros is True, a zero result has the sign of *X*. When *Towards* is zero, its sign has no bearing on the result. See A.5.3. 11

S'Aft For every fixed point subtype S: 12

13 S'Aft yields the number of decimal digits needed after the decimal point to accommodate the *delta* of the subtype S, unless the *delta* of the subtype S is greater than 0.1, in which case the attribute yields the value one. (S'Aft is the smallest positive integer N for which (10**N)*S'Delta is greater than or equal to one.) The value of this attribute is of the type *universal_integer*. See 3.5.10.

13.1/2 S'Alignment For every subtype S:

13.2/2 The value of this attribute is of type *universal_integer*, and nonnegative.

13.3/2 For an object X of subtype S, if S'Alignment is not zero, then X'Alignment is a nonzero integral multiple of S'Alignment unless specified otherwise by a representation item. See 13.3.

14/1 X'Alignment For a prefix X that denotes an object:

15/2 The value of this attribute is of type *universal_integer*, and nonnegative; zero means that the object is not necessarily aligned on a storage element boundary. If X'Alignment is not zero, then X is aligned on a storage unit boundary and X'Address is an integral multiple of X'Alignment (that is, the Address modulo the Alignment is zero).

16/2 See 13.3.

17 S'Base For every scalar subtype S:

18 S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the *base subtype* of the type. See 3.5.

19 S'Bit_Order For every specific record subtype S:

20 Denotes the bit ordering for the type of S. The value of this attribute is of type System.Bit_Order. See 13.5.3.

21/1 P'Body_Version
For a prefix P that statically denotes a program unit:

22 Yields a value of the predefined type String that identifies the version of the compilation unit that contains the body (but not any subunits) of the program unit. See E.3.

23 T'Callable For a prefix T that is of a task type (after any implicit dereference):

24 Yields the value True when the task denoted by T is *callable*, and False otherwise; See 9.9.

25 E'Caller For a prefix E that denotes an entry_declaration:

26/3 Yields a value of the type Task_Id that identifies the task whose call is now being serviced. Use of this attribute is allowed only inside an accept_statement, or entry_body after the entry_barrier, corresponding to the entry_declaration denoted by E. See C.7.1.

27 S'Ceiling For every subtype S of a floating point type *T*:

28 S'Ceiling denotes a function with the following specification:

29
```
function S'Ceiling (X : T)
  return T
```

30/5 The function yields the value $\lceil X \rceil$, that is, the smallest (most negative) integral value greater than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a negative sign when S'Signed_Zeros is True. See A.5.3.

30.1/6 S'Ceiling
For every subtype S of a fixed point type *T* that has integral values:

30.2/6 S'Ceiling denotes a function with the following specification:

```
function S'Ceiling (X : T)
   return T
```

30.3/6

The function yields the value *Ceiling(X)*, that is, the smallest (most negative) integral value greater than or equal to *X*. See A.5.4. 30.4/6

S'Class For every subtype S of a tagged type *T* (specific or class-wide): 31

S'Class denotes a subtype of the class-wide type (called *T*'Class in this document) for the class rooted at *T* (or if S already denotes a class-wide subtype, then S'Class is the same as S). 32

S'Class is unconstrained. However, if S is constrained, then the values of S'Class are only those that when converted to the type *T* belong to S. See 3.9. 33

S'Class For every subtype S of an untagged private type whose full view is tagged: 34

Denotes the class-wide subtype corresponding to the full view of S. This attribute is allowed only from the beginning of the private part in which the full view is declared, until the declaration of the full view. After the full view, the Class attribute of the full view can be used. See 7.3.1. 35

X'Component_Size 36/1

For a prefix X that denotes an array subtype or array object (after any implicit dereference):

Denotes the size in bits of components of the type of X. The value of this attribute is of type *universal_integer*. See 13.3. 37

S'Compose For every subtype S of a floating point type *T*: 38

S'Compose denotes a function with the following specification: 39

```
function S'Compose (Fraction : T;                                    40
                    Exponent : universal_integer)
  return T
```

Let *v* be the value $Fraction \cdot T\text{'Machine\_Radix}^{Exponent-k}$, where *k* is the normalized exponent of *Fraction*. If *v* is a machine number of the type *T*, or if $|v| \geq T$'Model_Small, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *Fraction* when S'Signed_Zeros is True. See A.5.3. 41

A'Constrained 42

For a prefix A that is of a discriminated type (after any implicit dereference):

Yields the value True if A denotes a constant, a value, a tagged object, or a constrained variable, and False otherwise. The value of this attribute is of the predefined type Boolean. See 3.7.2. 43/5

S'Copy_Sign For every subtype S of a floating point type *T*: 44

S'Copy_Sign denotes a function with the following specification: 45

```
function S'Copy_Sign (Value, Sign : T)                               46
  return T
```

If the value of *Value* is nonzero, the function yields a result whose magnitude is that of *Value* and whose sign is that of *Sign*; otherwise, it yields the value zero. Constraint_Error is optionally raised if the result is outside the base range of S. A zero result has the sign of *Sign* when S'Signed_Zeros is True. See A.5.3. 47

E'Count For a prefix E that denotes an entry of a task or protected unit: 48

Yields the number of calls presently queued on the entry E of the current instance of the unit. The value of this attribute is of the type *universal_integer*. See 9.9. 49

50/1 S'Definite For a prefix S that denotes a formal indefinite subtype:

51/3 S'Definite yields True if the actual subtype corresponding to S is definite; otherwise, it yields False. The value of this attribute is of the predefined type Boolean. See 12.5.1.

52 S'Delta For every fixed point subtype S:

53 S'Delta denotes the *delta* of the fixed point subtype S. The value of this attribute is of the type *universal_real*. See 3.5.10.

54 S'Denorm For every subtype S of a floating point type *T*:

55 Yields the value True if every value expressible in the form

$$\pm\ mantissa \cdot T\text{'Machine\_Radix}^{T\text{'Machine\_Emin}}$$

where *mantissa* is a nonzero *T*'Machine_Mantissa-digit fraction in the number base *T*'Machine_Radix, the first digit of which is zero, is a machine number (see 3.5.7) of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

56 S'Digits For every floating point subtype S:

57 S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. See 3.5.8.

58 S'Digits For every decimal fixed point subtype S:

59 S'Digits denotes the *digits* of the decimal fixed point subtype S, which corresponds to the number of decimal digits that are representable in objects of the subtype. The value of this attribute is of the type *universal_integer*. See 3.5.10.

59.1/5 S'Enum_Rep For every discrete subtype S:

59.2/5 S'Enum_Rep denotes a function with the following specification:

59.3/5
```
function S'Enum_Rep (Arg : S'Base) return universal_integer
```

59.4/5 This function returns the representation value of the value of Arg, as a value of type *universal_integer*. The *representation value* is the internal code specified in an enumeration representation clause, if any, for the type corresponding to the value of Arg, and otherwise is the position number of the value. See 13.4.

59.5/5 S'Enum_Val For every discrete subtype S:

59.6/5 S'Enum_Val denotes a function with the following specification:

59.7/5
```
function S'Enum_Val (Arg : universal_integer) return S'Base
```

59.8/5 This function returns a value of the type of S whose representation value equals the value of Arg. For the evaluation of a call on S'Enum_Val, if there is no value in the base range of its type with the given representation value, Constraint_Error is raised. See 13.4.

60 S'Exponent For every subtype S of a floating point type *T*:

61 S'Exponent denotes a function with the following specification:

62
```
function S'Exponent (X : T)
  return universal_integer
```

63 The function yields the normalized exponent of *X*. See A.5.3.

64 S'External_Tag
For every subtype S of a tagged type *T* (specific or class-wide):

65 S'External_Tag denotes an external string representation for S'Tag; it is of the predefined type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is implementation defined. See 13.13.2. See 13.3.

A'First

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 66/1

A'First denotes the lower bound of the first index range; its type is the corresponding index type. See 3.6.2. 67

S'First

For every scalar subtype S: 68

S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S. See 3.5. 69

A'First(N)

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 70/1

A'First(N) denotes the lower bound of the N-th index range; its type is the corresponding index type. See 3.6.2. 71

R.C'First_Bit 72

For a component C of a composite, non-array object R:

If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the first_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the first bit occupied by C. This offset is measured in bits. The first bit of a storage element is numbered zero. The value of this attribute is of the type *universal_integer*. See 13.5.2. 73/2

S'First_Valid 73.1/4

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S:

S'First_Valid denotes the smallest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S. See 3.5.5. 73.2/4

S'Floor

For every subtype S of a floating point type *T*: 74

S'Floor denotes a function with the following specification: 75

```
function S'Floor (X : T)
  return T
```
76

The function yields the value $\lfloor X \rfloor$, that is, the largest (most positive) integral value less than or equal to *X*. When *X* is zero, the result has the sign of *X*; a zero result otherwise has a positive sign. See A.5.3. 77/5

S'Floor 77.1/6

For every subtype S of a fixed point type *T* that has integral values:

S'Floor denotes a function with the following specification: 77.2/6

```
function S'Floor (X : T)
   return T
```
77.3/6

The function yields the value *Floor(X)*, that is, the largest (most positive) integral value less than or equal to *X*. See A.5.4. 77.4/6

S'Fore

For every fixed point subtype S: 78

S'Fore yields the minimum number of characters needed before the decimal point for the decimal representation of any value of the subtype S, assuming that the representation does not include an exponent, but includes a one-character prefix that is either a minus sign or a space. (This minimum number does not include superfluous zeros or underlines, and is at least 2.) The value of this attribute is of the type *universal_integer*. See 3.5.10. 79

S'Fraction

For every subtype S of a floating point type *T*: 80

81 S'Fraction denotes a function with the following specification:

82
```
function S'Fraction (X : T)
  return T
```

83 The function yields the value $X \cdot T'\text{Machine\_Radix}^{-k}$, where k is the normalized exponent of X. A zero result, which can only occur when X is zero, has the sign of X. See A.5.3.

83.1/3 X'Has_Same_Storage

For a prefix X that denotes an object:

83.2/4 X'Has_Same_Storage denotes a function with the following specification:

83.3/3
```
function X'Has_Same_Storage (Arg : any_type)
  return Boolean
```

83.4/4 The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved. It returns True if the representation of the object denoted by the actual parameter occupies exactly the same bits as the representation of the object denoted by X and the objects occupy at least one bit; otherwise, it returns False. See 13.3.

84/1 E'Identity For a prefix E that denotes an exception:

85 E'Identity returns the unique identity of the exception. The type of this attribute is Exception_Id. See 11.4.1.

86 T'Identity For a prefix T that is of a task type (after any implicit dereference):

87 Yields a value of the type Task_Id that identifies the task denoted by T. See C.7.1.

88 S'Image For every subtype S of a type T:

89/5 S'Image denotes a function with the following specification:

90/5
```
function S'Image(Arg : S'Base)
  return String
```

91/5 S'Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Get. See 4.10.

91.1/5 X'Image For a prefix X of a type T other than *universal_real* or *universal_fixed*:

91.2/5 X'Image denotes the result of calling function S'Image with *Arg* being X, where S is the nominal subtype of X. See 4.10.

91.3/5 E'Index For a prefix E that denotes an entry declaration of an entry family:

91.4/5 Within a precondition or postcondition expression for entry family E, denotes the value of the entry index for the call of E. The nominal subtype of this attribute is the entry index subtype. See 6.1.1.

92 S'Class'Input

For every subtype S'Class of a class-wide type *T*'Class:

93 S'Class'Input denotes a function with the following specification:

94/2
```
function S'Class'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T'Class
```

95/5 First reads the external tag from *Stream* and determines the corresponding internal tag (by calling Tags.Descendant_Tag(String'Input(*Stream*), S'Tag) which can raise Tag_Error — see 3.9) and then dispatches to the subprogram denoted by the Input attribute of the specific type identified by the internal tag; returns that result. If the specific type identified by the internal tag is abstract, Constraint_Error is raised. See 13.13.2.

S'Input

For every subtype S of a specific type *T*: 96

S'Input denotes a function with the following specification: 97

```
function S'Input(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class)
   return T
```
98/2

S'Input reads and returns one value from *Stream*, using any bounds or discriminants written by a corresponding S'Output to determine how much to read. See 13.13.2. 99

A'Last

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 100/1

A'Last denotes the upper bound of the first index range; its type is the corresponding index type. See 3.6.2. 101

S'Last

For every scalar subtype S: 102

S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S. See 3.5. 103

A'Last(N)

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 104/1

A'Last(N) denotes the upper bound of the N-th index range; its type is the corresponding index type. See 3.6.2. 105

R.C'Last_Bit 106

For a component C of a composite, non-array object R:

If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the last_bit of the component_clause; otherwise, denotes the offset, from the start of the first of the storage elements occupied by C, of the last bit occupied by C. This offset is measured in bits. The value of this attribute is of the type *universal_integer*. See 13.5.2. 107/2

S'Last_Valid 107.1/4

For every static discrete subtype S for which there exists at least one value belonging to S that satisfies the predicates of S:

S'Last_Valid denotes the largest value that belongs to S and satisfies the predicates of S. The value of this attribute is of the type of S. See 3.5.5. 107.2/4

S'Leading_Part 108

For every subtype S of a floating point type *T*:

S'Leading_Part denotes a function with the following specification: 109

```
function S'Leading_Part (X : T;
                         Radix_Digits : universal_integer)
  return T
```
110

Let *v* be the value *T*'Machine_Radix$^{k-Radix_Digits}$, where *k* is the normalized exponent of *X*. The function yields the value 111

- $\lfloor X/v \rfloor \cdot v$, when *X* is nonnegative and *Radix_Digits* is positive; 112
- $\lceil X/v \rceil \cdot v$, when *X* is negative and *Radix_Digits* is positive. 113

Constraint_Error is raised when *Radix_Digits* is zero or negative. A zero result, which can only occur when *X* is zero, has the sign of *X*. See A.5.3. 114

A'Length

For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 115/1

116 A'Length denotes the number of values of the first index range (zero for a null range); its type is *universal_integer*. See 3.6.2.

117/1 A'Length(N) For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype:

118 A'Length(N) denotes the number of values of the N-th index range (zero for a null range); its type is *universal_integer*. See 3.6.2.

119 S'Machine For every subtype S of a floating point type *T*:

120 S'Machine denotes a function with the following specification:

121
```
function S'Machine (X : T)
  return T
```

122 If *X* is a machine number of the type *T*, the function yields *X*; otherwise, it yields the value obtained by rounding or truncating *X* to either one of the adjacent machine numbers of the type *T*. Constraint_Error is raised if rounding or truncating *X* to the precision of the machine numbers results in a value outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

123 S'Machine_Emax
For every subtype S of a floating point type *T*:

124 Yields the largest (most positive) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

125 S'Machine_Emin
For every subtype S of a floating point type *T*:

126 Yields the smallest (most negative) value of *exponent* such that every value expressible in the canonical form (for the type *T*), having a *mantissa* of *T*'Machine_Mantissa digits, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

127 S'Machine_Mantissa
For every subtype S of a floating point type *T*:

128 Yields the largest value of *p* such that every value expressible in the canonical form (for the type *T*), having a *p*-digit *mantissa* and an *exponent* between *T*'Machine_Emin and *T*'Machine_Emax, is a machine number (see 3.5.7) of the type *T*. This attribute yields a value of the type *universal_integer*. See A.5.3.

129 S'Machine_Overflows
For every subtype S of a floating point type *T*:

130 Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3.

131 S'Machine_Overflows
For every subtype S of a fixed point type *T*:

132 Yields the value True if overflow and divide-by-zero are detected and reported by raising Constraint_Error for every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4.

S'Machine_Radix 133

For every subtype S of a floating point type *T*:

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*. See A.5.3. 134

S'Machine_Radix 135

For every subtype S of a fixed point type *T*:

Yields the radix of the hardware representation of the type *T*. The value of this attribute is of the type *universal_integer*. See A.5.4. 136

S'Machine_Rounding 136.1/2

For every subtype S of a floating point type *T*:

S'Machine_Rounding denotes a function with the following specification: 136.2/2

```
function S'Machine_Rounding (X : T)                    136.3/2
  return T
```

The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. A zero result has the sign of *X* when S'Signed_Zeros is True. This function provides access to the rounding behavior which is most efficient on the target processor. See A.5.3. 136.4/2

S'Machine_Rounding 136.5/6

For every subtype S of a fixed point type *T* that has integral values:

S'Machine_Rounding denotes a function with the following specification: 136.6/6

```
function S'Machine_Rounding (X : T)                    136.7/6
   return T
```

The function yields the integral value nearest to *X*. If *X* lies exactly halfway between two integers, one of those integers is returned, but which of them is returned is unspecified. This function provides access to the rounding behavior which is most efficient on the target processor. See A.5.4. 136.8/6

S'Machine_Rounds 137

For every subtype S of a floating point type *T*:

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3. 138

S'Machine_Rounds 139

For every subtype S of a fixed point type *T*:

Yields the value True if rounding is performed on inexact results of every predefined operation that yields a result of the type *T*; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.4. 140

S'Max For every scalar subtype S: 141

S'Max denotes a function with the following specification: 142

```
function S'Max(Left, Right : S'Base)                   143
  return S'Base
```

The function returns the greater of the values of the two parameters. See 3.5. 144

S'Max_Alignment_For_Allocation 144.1/3

For every subtype S:

144.2/5 Denotes the maximum value for Alignment that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*. See 13.11.1.

145 S'Max_Size_In_Storage_Elements
For every subtype S:

146/5 Denotes the maximum value for Size_In_Storage_Elements that can be requested by the implementation via Allocate for an access type whose designated subtype is S. The value of this attribute is of type *universal_integer*. See 13.11.1.

147 S'Min For every scalar subtype S:

148 S'Min denotes a function with the following specification:

149
```
function S'Min(Left, Right : S'Base)
  return S'Base
```

150 The function returns the lesser of the values of the two parameters. See 3.5.

150.1/2 S'Mod For every modular subtype S:

150.2/2 S'Mod denotes a function with the following specification:

150.3/2
```
function S'Mod (Arg : universal_integer)
  return S'Base
```

150.4/2 This function returns *Arg* **mod** S'Modulus, as a value of the type of S. See 3.5.4.

151 S'Model For every subtype S of a floating point type *T*:

152 S'Model denotes a function with the following specification:

153
```
function S'Model (X : T)
  return T
```

154 If the Numerics Annex is not supported, the meaning of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. See A.5.3.

155 S'Model_Emin
For every subtype S of a floating point type *T*:

156 If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to the value of *T*'Machine_Emin. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*. See A.5.3.

157 S'Model_Epsilon
For every subtype S of a floating point type *T*:

158 Yields the value $T\text{'Machine\_Radix}^{1 - T\text{'Model\_Mantissa}}$. The value of this attribute is of the type *universal_real*. See A.5.3.

159 S'Model_Mantissa
For every subtype S of a floating point type *T*:

160 If the Numerics Annex is not supported, this attribute yields an implementation defined value that is greater than or equal to $\lceil d \cdot \log(10) / \log(T\text{'Machine\_Radix}) \rceil + 1$, where d is the requested decimal precision of *T*, and less than or equal to the value of *T*'Machine_Mantissa. See G.2.2 for further requirements that apply to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_integer*. See A.5.3.

161 S'Model_Small
For every subtype S of a floating point type *T*:

Yields the value T'Machine_Radix$^{T\text{'Model_Emin} - 1}$. The value of this attribute is of the type *universal_real*. See A.5.3. 162

S'Modulus For every modular subtype S: 163

S'Modulus yields the modulus of the type of S, as a value of the type *universal_integer*. See 3.5.4. 164

S'Object_Size 164.1/5

For every subtype S:

If S is definite, denotes the size (in bits) of a stand-alone aliased object, or a component of subtype S in the absence of an aspect_specification or representation item that specifies the size of the object or component. If S is indefinite, the meaning is implementation-defined. The value of this attribute is of the type *universal_integer*. See 13.3. 164.2/5

X'Old For a prefix X that denotes an object of a nonlimited type: 164.3/3

Each X'Old in a postcondition expression that is enabled, other than those that occur in subexpressions that are determined to be unevaluated, denotes a constant that is implicitly declared at the beginning of the subprogram body, entry body, or accept statement. See 6.1.1. 164.4/5

S'Class'Output 165

For every subtype S'Class of a class-wide type T'Class:

S'Class'Output denotes a procedure with the following specification: 166

```
procedure S'Class'Output(                                                    167/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```

First writes the external tag of *Item* to *Stream* (by calling String'Output(*Stream*, Tags.-External_Tag(*Item*'Tag)) — see 3.9) and then dispatches to the subprogram denoted by the Output attribute of the specific type identified by the tag. Tag_Error is raised if the tag of Item identifies a type declared at an accessibility level deeper than that of S. See 13.13.2. 168/2

S'Output For every subtype S of a specific type T: 169

S'Output denotes a procedure with the following specification: 170

```
procedure S'Output(                                                          171/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```

S'Output writes the value of *Item* to *Stream*, including any bounds or discriminants. See 13.13.2. 172

X'Overlaps_Storage 172.1/3

For a prefix X that denotes an object:

X'Overlaps_Storage denotes a function with the following specification: 172.2/3

```
function X'Overlaps_Storage (Arg : any_type)                                 172.3/3
  return Boolean
```

The actual parameter shall be a name that denotes an object. The object denoted by the actual parameter can be of any type. This function evaluates the names of the objects involved and returns True if the representation of the object denoted by the actual parameter shares at least one bit with the representation of the object denoted by X; otherwise, it returns False. See 13.3. 172.4/3

X'Parallel_Reduce(Reducer, Initial_Value) 172.5/5

For a prefix X of an array type (after any implicit dereference), or that denotes an iterable container object (see 5.5.1):

172.6/5 X'Parallel_Reduce is a reduction expression that yields a result equivalent to replacing the attribute identifier with Reduce and the prefix of the attribute with the value_sequence:

172.7/5
```
[parallel for Item of X => Item]
```

172.8/5 See 4.5.10.

173/1 D'Partition_Id

For a prefix D that denotes a library-level declaration, excepting a declaration of or within a declared-pure library unit:

174 Denotes a value of the type *universal_integer* that identifies the partition in which D was elaborated. If D denotes the declaration of a remote call interface library unit (see E.2.3) the given partition is the one where the body of D was elaborated. See E.1.

175 S'Pos For every discrete subtype S:

176 S'Pos denotes a function with the following specification:

177
```
function S'Pos(Arg : S'Base)
  return universal_integer
```

178 This function returns the position number of the value of *Arg*, as a value of type *universal_integer*. See 3.5.5.

179 R.C'Position For a component C of a composite, non-array object R:

180/2 If the nondefault bit ordering applies to the composite type, and if a component_clause specifies the placement of C, denotes the value given for the position of the component_clause; otherwise, denotes the same value as R.C'Address – R'Address. The value of this attribute is of the type *universal_integer*. See 13.5.2.

181 S'Pred For every scalar subtype S:

182 S'Pred denotes a function with the following specification:

183
```
function S'Pred(Arg : S'Base)
  return S'Base
```

184 For an enumeration type, the function returns the value whose position number is one less than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of subtracting one from the value of *Arg*. For a fixed point type, the function returns the result of subtracting *small* from the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately below the value of *Arg*; Constraint_Error is raised if there is no such machine number. See 3.5.

184.1/5 S'Preelaborable_Initialization

For a nonformal composite subtype S declared within the visible part of a package or a generic package, or a generic formal private subtype or formal derived subtype:

184.2/6 This attribute is of Boolean type, and its value reflects whether the type of S has preelaborable initialization. See 10.2.1.

184.3/2 P'Priority For a prefix P that denotes a protected object:

184.4/2 Denotes a non-aliased component of the protected object P. This component is of type System.Any_Priority and its value is the priority of P. P'Priority denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P. See D.5.2.

184.5/5 S'Put_Image For every subtype S of a type T other than *universal_real* or *universal_fixed*:

184.6/5 S'Put_Image denotes a procedure with the following specification:

```
procedure S'Put_Image                                                    184.7/5
   (Buffer : in out
                Ada.Strings.Text_Buffers.Root_Buffer_Type'Class;
    Arg    : in T);
```

The default implementation of S'Put_Image writes (using Wide_Wide_Put) an *image* of the value of *Arg*. See 4.10. 184.8/5

A'Range For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 185/1

A'Range is equivalent to the range A'First .. A'Last, except that the prefix A is only evaluated once. See 3.6.2. 186

S'Range For every scalar subtype S: 187

S'Range is equivalent to the range S'First .. S'Last. See 3.5. 188

A'Range(N) For a prefix A that is of an array type (after any implicit dereference), or denotes a constrained array subtype: 189/1

A'Range(N) is equivalent to the range A'First(N) .. A'Last(N), except that the prefix A is only evaluated once. See 3.6.2. 190

S'Class'Read For every subtype S'Class of a class-wide type *T*'Class: 191

S'Class'Read denotes a procedure with the following specification: 192

```
procedure S'Class'Read(                                                  193/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T'Class)
```

Dispatches to the subprogram denoted by the Read attribute of the specific type identified by the tag of Item. See 13.13.2. 194

S'Read For every subtype S of a specific type *T*: 195

S'Read denotes a procedure with the following specification: 196

```
procedure S'Read(                                                        197/2
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : out T)
```

S'Read reads the value of *Item* from *Stream*. See 13.13.2. 198

V'Reduce(Reducer, Initial_Value) 198.1/5

For a value_sequence V:

This attribute represents a *reduction expression*, and is in the form of a reduction_attribute_reference. See 4.5.10. 198.2/5

X'Reduce(Reducer, Initial_Value) 198.3/5

For a prefix X of an array type (after any implicit dereference), or that denotes an iterable container object (see 5.5.1):

X'Reduce is a reduction expression that yields a result equivalent to replacing the prefix of the attribute with the value_sequence: 198.4/5

```
[for Item of X => Item]                                                  198.5/5
```

See 4.5.10. 198.6/5

P'Relative_Deadline 198.7/5

For a prefix P that denotes a protected object:

Denotes a non-aliased component of the protected object P. This component is of type Ada.Real_Time.Time_Span and its value is the relative deadline of P. P'Relative_Deadline 198.8/5

denotes a variable if and only if P denotes a variable. A reference to this attribute shall appear only within the body of P. See D.5.2.

199 S'Remainder For every subtype S of a floating point type *T*:

200 S'Remainder denotes a function with the following specification:

201
```
function S'Remainder (X, Y : T)
  return T
```

202 For nonzero *Y*, let *v* be the value $X - n \cdot Y$, where *n* is the integer nearest to the exact value of *X*/*Y*; if $|n - X/Y| = 1/2$, then *n* is chosen to be even. If *v* is a machine number of the type *T*, the function yields *v*; otherwise, it yields zero. Constraint_Error is raised if *Y* is zero. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

202.1/3 F'Result For a prefix F that denotes a function declaration or an access-to-function type:

202.2/5 Within a postcondition expression for F, denotes the return object of the function call for which the postcondition expression is evaluated. The type of this attribute is that of the result subtype of the function or access-to-function type except within a Post'Class postcondition expression for a function with a controlling result or with a controlling access result; in those cases the type of the attribute is described above as part of the Name Resolution Rules for Post'Class. See 6.1.1.

203 S'Round For every decimal fixed point subtype S:

204 S'Round denotes a function with the following specification:

205
```
function S'Round(X : universal_real)
  return S'Base
```

206 The function returns the value obtained by rounding X (away from 0, if X is midway between two values of the type of S). See 3.5.10.

207 S'Rounding For every subtype S of a floating point type *T*:

208 S'Rounding denotes a function with the following specification:

209
```
function S'Rounding (X : T)
  return T
```

210 The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

210.1/6 S'Rounding

For every subtype S of a fixed point type *T* that has integral values:

210.2/6 S'Rounding denotes a function with the following specification:

210.3/6
```
function S'Rounding (X : T)
   return T
```

210.4/6 The function yields the integral value nearest to *X*, rounding away from zero if *X* lies exactly halfway between two integers. See A.5.4.

211 S'Safe_First

For every subtype S of a floating point type *T*:

212 Yields the lower bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*. See A.5.3.

213 S'Safe_Last For every subtype S of a floating point type *T*:

Yields the upper bound of the safe range (see 3.5.7) of the type *T*. If the Numerics Annex is not supported, the value of this attribute is implementation defined; see G.2.2 for the definition that applies to implementations supporting the Numerics Annex. The value of this attribute is of the type *universal_real*. See A.5.3. 214

S'Scale For every decimal fixed point subtype S: 215

S'Scale denotes the *scale* of the subtype S, defined as the value N such that S'Delta = 10.0**(–N). The scale indicates the position of the point relative to the rightmost significant digits of values of subtype S. The value of this attribute is of the type *universal_integer*. See 3.5.10. 216

S'Scaling For every subtype S of a floating point type *T*: 217

S'Scaling denotes a function with the following specification: 218

```
function S'Scaling (X : T;                                    219
                    Adjustment : universal_integer)
  return T
```

Let *v* be the value $X \cdot T\text{'Machine\_Radix}^{Adjustment}$. If *v* is a machine number of the type *T*, or if $|v| \geq T\text{'Model\_Small}$, the function yields *v*; otherwise, it yields either one of the machine numbers of the type *T* adjacent to *v*. Constraint_Error is optionally raised if *v* is outside the base range of S. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3. 220

S'Signed_Zeros 221

For every subtype S of a floating point type *T*:

Yields the value True if the hardware representation for the type *T* has the capability of representing both positively and negatively signed zeros, these being generated and used by the predefined operations of the type *T* as specified in IEC 559:1989; yields the value False otherwise. The value of this attribute is of the predefined type Boolean. See A.5.3. 222

S'Size For every subtype S: 223

If S is definite, denotes the size (in bits) that the implementation would choose for the following objects of subtype S: 224

- A record component of subtype S when the record type is packed. 225
- The formal parameter of an instance of Unchecked_Conversion that converts from subtype S to some other subtype. 226

If S is indefinite, the meaning is implementation defined. The value of this attribute is of the type *universal_integer*. See 13.3. 227

X'Size For a prefix X that denotes an object: 228/1

Denotes the size in bits of the representation of the object. The value of this attribute is of the type *universal_integer*. See 13.3. 229

S'Small For every fixed point subtype S: 230

S'Small denotes the *small* of the type of S. The value of this attribute is of the type *universal_real*. See 3.5.10. 231

S'Storage_Pool 232

For every access-to-object subtype S:

Denotes the storage pool of the type of S. The type of this attribute is Root_Storage_-Pool'Class. See 13.11. 233

S'Storage_Size 234

For every access-to-object subtype S:

235 Yields the result of calling Storage_Size(S'Storage_Pool), which is intended to be a measure of the number of storage elements reserved for the pool. The type of this attribute is *universal_integer*. See 13.11.

236/1 T'Storage_Size

For a prefix T that denotes a task object (after any implicit dereference):

237 Denotes the number of storage elements reserved for the task. The value of this attribute is of the type *universal_integer*. The Storage_Size includes the size of the task's stack, if any. The language does not specify whether or not it includes other storage associated with the task (such as the "task control block" used by some implementations.) See 13.3.

237.1/2 S'Stream_Size

For every subtype S of an elementary type *T*:

237.2/3 Denotes the number of bits read from or written to a stream by the default implementations of S'Read and S'Write. Hence, the number of stream elements required per item of elementary type *T* is:

237.3/2
```
T'Stream_Size / Ada.Streams.Stream_Element'Size
```

237.4/2 The value of this attribute is of type *universal_integer* and is a multiple of Stream_Element'Size. See 13.13.2.

238 S'Succ For every scalar subtype S:

239 S'Succ denotes a function with the following specification:

240
```
function S'Succ(Arg : S'Base)
  return S'Base
```

241 For an enumeration type, the function returns the value whose position number is one more than that of the value of *Arg*; Constraint_Error is raised if there is no such value of the type. For an integer type, the function returns the result of adding one to the value of *Arg*. For a fixed point type, the function returns the result of adding *small* to the value of *Arg*. For a floating point type, the function returns the machine number (as defined in 3.5.7) immediately above the value of *Arg*; Constraint_Error is raised if there is no such machine number. See 3.5.

242 S'Tag For every subtype S of a tagged type *T* (specific or class-wide):

243 S'Tag denotes the tag of the type *T* (or if *T* is class-wide, the tag of the root type of the corresponding class). The value of this attribute is of type Tag. See 3.9.

244 X'Tag For a prefix X that is of a class-wide tagged type (after any implicit dereference):

245 X'Tag denotes the tag of X. The value of this attribute is of type Tag. See 3.9.

246 T'Terminated For a prefix T that is of a task type (after any implicit dereference):

247 Yields the value True if the task denoted by T is terminated, and False otherwise. The value of this attribute is of the predefined type Boolean. See 9.9.

248 S'Truncation For every subtype S of a floating point type *T*:

249 S'Truncation denotes a function with the following specification:

250
```
function S'Truncation (X : T)
  return T
```

251 The function yields the value $\lceil X \rceil$ when *X* is negative, and $\lfloor X \rfloor$ otherwise. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3.

251.1/6 S'Truncation

For every subtype S of a fixed point type *T* that has integral values:

S'Truncation denotes a function with the following specification: 251.2/6

```
function S'Truncation (X : T)                                    251.3/6
   return T
```

The function yields the value *Ceiling(X)* when *X* is negative, and *Floor(X)* otherwise. See A.5.4. 251.4/6

S'Unbiased_Rounding 252

For every subtype S of a floating point type *T*:

S'Unbiased_Rounding denotes a function with the following specification: 253

```
function S'Unbiased_Rounding (X : T)                             254
  return T
```

The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. A zero result has the sign of *X* when S'Signed_Zeros is True. See A.5.3. 255

S'Unbiased_Rounding 255.1/6

For every subtype S of a fixed point type *T* that has integral values:

S'Unbiased_Rounding denotes a function with the following specification: 255.2/6

```
function S'Unbiased_Rounding (X : T)                             255.3/6
   return T
```

The function yields the integral value nearest to *X*, rounding toward the even integer if *X* lies exactly halfway between two integers. See A.5.4. 255.4/6

X'Unchecked_Access 256

For a prefix X that denotes an aliased view of an object:

All rules and semantics that apply to X'Access (see 3.10.2) apply also to X'Unchecked_Access, except that, for the purposes of accessibility rules and checks, it is as if X were declared immediately within a library package. See 13.10. 257

S'Val For every discrete subtype S: 258

S'Val denotes a function with the following specification: 259

```
function S'Val(Arg : universal_integer)                          260
  return S'Base
```

This function returns a value of the type of S whose position number equals the value of *Arg*. See 3.5.5. 261

X'Valid For a prefix X that denotes a scalar object (after any implicit dereference): 262

Yields True if and only if the object denoted by X is normal, has a valid representation, and then, if the preceding conditions hold, the value of X also satisfies the predicates of the nominal subtype of X. The value of this attribute is of the predefined type Boolean. See 13.9.2. 263/4

S'Value For every scalar subtype S: 264

S'Value denotes a function with the following specification: 265

```
function S'Value(Arg : String)                                   266
  return S'Base
```

This function returns a value given an image of the value as a String, ignoring any leading or trailing spaces. See 3.5. 267

P'Version For a prefix P that statically denotes a program unit: 268/1

Yields a value of the predefined type String that identifies the version of the compilation unit that contains the declaration of the program unit. See E.3. 269

270 S'Wide_Image

For every subtype S of a type T:

271/5 S'Wide_Image denotes a function with the following specification:

272/5
```
function S'Wide_Image(Arg : S'Base)
  return Wide_String
```

273/5 S'Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Wide_Get. See 4.10.

273.1/5 X'Wide_Image

For a prefix X of a type T other than *universal_real* or *universal_fixed*:

273.2/5 X'Wide_Image denotes the result of calling function S'Wide_Image with *Arg* being X, where S is the nominal subtype of X. See 4.10.

274 S'Wide_Value

For every scalar subtype S:

275 S'Wide_Value denotes a function with the following specification:

276
```
function S'Wide_Value(Arg : Wide_String)
  return S'Base
```

277 This function returns a value given an image of the value as a Wide_String, ignoring any leading or trailing spaces. See 3.5.

277.1/2 S'Wide_Wide_Image

For every subtype S of a type T:

277.2/5 S'Wide_Wide_Image denotes a function with the following specification:

277.3/5
```
function S'Wide_Wide_Image(Arg : S'Base)
  return Wide_Wide_String
```

277.4/5 S'Wide_Wide_Image calls S'Put_Image passing *Arg* (which will typically store a sequence of character values in a text buffer) and then returns the result of retrieving the contents of that buffer with function Wide_Wide_Get. See 4.10.

277.5/5 X'Wide_Wide_Image

For a prefix X of a type T other than *universal_real* or *universal_fixed*:

277.6/5 X'Wide_Wide_Image denotes the result of calling function S'Wide_Wide_Image with *Arg* being X, where S is the nominal subtype of X. See 4.10.

277.7/2 S'Wide_Wide_Value

For every scalar subtype S:

277.8/2 S'Wide_Wide_Value denotes a function with the following specification:

277.9/2
```
function S'Wide_Wide_Value(Arg : Wide_Wide_String)
  return S'Base
```

277.10/2 This function returns a value given an image of the value as a Wide_Wide_String, ignoring any leading or trailing spaces. See 3.5.

277.11/2 S'Wide_Wide_Width

For every scalar subtype S:

277.12/5 S'Wide_Wide_Width denotes the maximum length of a Wide_Wide_String returned by S'Wide_Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5.

S'Wide_Width 278

For every scalar subtype S:

S'Wide_Width denotes the maximum length of a Wide_String returned by S'Wide_Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5. 279/5

S'Width For every scalar subtype S: 280

S'Width denotes the maximum length of a String returned by S'Image over all values of the subtype S, assuming a default implementation of S'Put_Image. It denotes zero for a subtype that has a null range. Its type is *universal_integer*. See 3.5. 281/5

S'Class'Write 282

For every subtype S'Class of a class-wide type *T*'Class:

S'Class'Write denotes a procedure with the following specification: 283

```
procedure S'Class'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item   : in T'Class)
```
284/2

Dispatches to the subprogram denoted by the Write attribute of the specific type identified by the tag of Item. See 13.13.2. 285

S'Write For every subtype S of a specific type *T*: 286

S'Write denotes a procedure with the following specification: 287

```
procedure S'Write(
   Stream : not null access Ada.Streams.Root_Stream_Type'Class;
   Item : in T)
```
288/2

S'Write writes the value of *Item* to *Stream*. See 13.13.2. 289

Annex L
(informative)
Language-Defined Pragmas

This Annex summarizes the definitions given elsewhere of the language-defined pragmas. 1

pragma Admission_Policy (*policy*_identifier); — See D.4.1. 1.1/5

This paragraph was deleted. 2/5

pragma All_Calls_Remote[(*library_unit*_name)]; — See J.15.15. 2.1/5

pragma Assert([Check =>] *boolean*_expression[, [Message =>] *string*_expression]); — See 11.4.2. 2.2/2

pragma Assertion_Policy(*policy*_identifier); — See 11.4.2. 2.3/2

pragma Assertion_Policy(2.4/3
*assertion*_aspect_mark => *policy*_identifier
{, *assertion*_aspect_mark => *policy*_identifier}); — See 11.4.2.

This paragraph was deleted. 3/3

pragma Asynchronous (local_name); — See J.15.13. 3.1/3

This paragraph was deleted. 4/3

pragma Atomic (local_name); — See J.15.8. 4.1/3

This paragraph was deleted. 5/3

pragma Atomic_Components (*array*_local_name); — See J.15.8. 5.1/3

This paragraph was deleted. 6/3

pragma Attach_Handler (*handler*_name, expression); — See J.15.7. 6.1/3

pragma Conflict_Check_Policy (*policy*_identifier[, *policy*_identifier]); — See 9.10.1. 6.2/5

This paragraph was deleted. 7/3

This paragraph was deleted. 8/3

pragma Convention([Convention =>] *convention*_identifier,[Entity =>] local_name); — See J.15.5. 8.1/3

pragma CPU (expression); — See J.15.9. 8.2/3

pragma Default_Storage_Pool (storage_pool_indicator); — See 13.11.3. 8.3/3

pragma Detect_Blocking; — See H.5. 8.4/2

pragma Discard_Names[([On =>] local_name)]; — See C.5. 9

pragma Dispatching_Domain (expression); — See J.15.10. 9.1/3

pragma Elaborate(*library_unit*_name{, *library_unit*_name}); — See 10.2.1. 10

pragma Elaborate_All(*library_unit*_name{, *library_unit*_name}); — See 10.2.1. 11

This paragraph was deleted. 12/5

12.1/5 **pragma** Elaborate_Body[(*library_unit*_name)]; — See J.15.14.

13/3 *This paragraph was deleted.*

13.1/3 **pragma** Export(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]); — See J.15.5.

13.2/5 **pragma** Generate_Deadlines; — See D.2.6.

14/3 *This paragraph was deleted.*

14.1/3 **pragma** Import(
[Convention =>] *convention*_identifier, [Entity =>] local_name
[, [External_Name =>] *external_name_string*_expression]
[, [Link_Name =>] *link_name_string*_expression]); — See J.15.5.

14.2/3 **pragma** Independent (*component*_local_name); — See J.15.8.

14.3/3 **pragma** Independent_Components (local_name); — See J.15.8.

15/3 *This paragraph was deleted.*

15.1/3 **pragma** Inline (name{, name}); — See J.15.1.

16 **pragma** Inspection_Point[(*object*_name {, *object*_name})]; — See H.3.2.

17/3 *This paragraph was deleted.*

17.1/3 **pragma** Interrupt_Handler (*handler*_name); — See J.15.7.

18/3 *This paragraph was deleted.*

18.1/3 **pragma** Interrupt_Priority [(expression);] — See J.15.11.

19 **pragma** Linker_Options(*string*_expression); — See B.1.

20 **pragma** List(identifier); — See 2.8.

21 **pragma** Locking_Policy(*policy*_identifier); — See D.3.

21.1/3 *This paragraph was deleted.*

21.2/3 **pragma** No_Return (*subprogram*_local_name{, *subprogram*_local_name}); — See J.15.2.

22 **pragma** Normalize_Scalars; — See H.1.

23 **pragma** Optimize(identifier); — See 2.8.

24/3 *This paragraph was deleted.*

24.1/3 **pragma** Pack (*first_subtype*_local_name); — See J.15.3.

25 **pragma** Page; — See 2.8.

25.1/2 **pragma** Partition_Elaboration_Policy (*policy*_identifier); — See H.6.

25.2/5 *This paragraph was deleted.*

25.3/5 **pragma** Preelaborable_Initialization(direct_name); — See J.15.14.

This paragraph was deleted. 26/5

pragma Preelaborate[(*library_unit*_name)]; — See J.15.14. 26.1/5

This paragraph was deleted. 27/3

pragma Priority (expression); — See J.15.11. 27.1/3

pragma Priority_Specific_Dispatching (27.2/2
*policy*_identifier, *first_priority*_expression, *last_priority*_expression); — See D.2.2.

pragma Profile (*profile*_identifier {, *profile*_pragma_argument_association}); — See 13.12. 27.3/3

This paragraph was deleted. 27.4/3

This paragraph was deleted. 28/5

pragma Pure[(*library_unit*_name)]; — See J.15.14. 28.1/5

pragma Queuing_Policy(*policy*_identifier); — See D.4. 29

This paragraph was deleted. 29.1/3

pragma Relative_Deadline (*relative_deadline*_expression); — See J.15.12. 29.2/3

This paragraph was deleted. 30/5

pragma Remote_Call_Interface[(*library_unit*_name)]; — See J.15.15. 30.1/5

This paragraph was deleted. 31/5

pragma Remote_Types[(*library_unit*_name)]; — See J.15.15. 31.1/5

pragma Restrictions(restriction{, restriction}); — See 13.12. 32

pragma Reviewable; — See H.3.1. 33

This paragraph was deleted. 34/5

pragma Shared_Passive[(*library_unit*_name)]; — See J.15.15. 34.1/5

This paragraph was deleted. 35/3

pragma Storage_Size (expression); — See J.15.4. 35.1/3

pragma Suppress(identifier); — See 11.5. 36

pragma Task_Dispatching_Policy(*policy*_identifier); — See D.2.2. 37

This paragraph was deleted. 37.1/3

pragma Unchecked_Union (*first_subtype*_local_name); — See J.15.6. 37.2/3

pragma Unsuppress(identifier); — See 11.5. 37.3/2

This paragraph was deleted. 38/3

pragma Volatile (local_name); — See J.15.8. 38.1/3

This paragraph was deleted. 39/3

pragma Volatile_Components (*array*_local_name); — See J.15.8. 39.1/3

Annex M
(informative)
Summary of Documentation Requirements

The Ada language allows for certain target machine dependences in a controlled manner. Each Ada implementation is required to document many characteristics and properties of the target system. This Reference Manual contains specific documentation requirements. In addition, many characteristics that require documentation are identified throughout this Reference Manual as being implementation defined. Finally, this Reference Manual requires documentation of whether implementation advice is followed. The following subclauses provide summaries of these documentation requirements. 1/5

M.1 Specific Documentation Requirements

In addition to implementation-defined characteristics, each Ada implementation is required to document various properties of the implementation: 1/5

- The behavior of implementations in implementation-defined situations shall be documented — see M.2 for a listing. See 1.1.3(19). 2/2
- The set of values that a user-defined Allocate procedure needs to accept for the Alignment parameter. How the standard storage pool is chosen, and how storage is allocated by standard storage pools. See 13.11(22/5). 3/2
- The algorithm used for random number generation, including a description of its period. See A.5.2(44). 4/2
- The minimum time interval between calls to the time-dependent Reset procedure that is guaranteed to initiate different random number sequences. See A.5.2(45). 5/2
- The conditions under which Io_Exceptions.Name_Error, Io_Exceptions.Use_Error, and Io_Exceptions.Device_Error are propagated. See A.13(15). 6/2
- The behavior of package Environment_Variables when environment variables are changed by external mechanisms. See A.17(30/2). 7/2
- The overhead of calling machine-code or intrinsic subprograms. See C.1(6). 8/2
- The types and attributes used in machine code insertions. See C.1(7). 9/2
- The subprogram calling conventions for all supported convention identifiers. See C.1(8/3). 10/2
- The mapping between the Link_Name or Ada designator and the external link name. See C.1(9). 11/2
- The treatment of interrupts. See C.3(22). 12/2
- The metrics for interrupt handlers. See C.3.1(16). 13/2
- If the Ceiling_Locking policy is in effect, the default ceiling priority for a protected object that specifies an interrupt handler aspect. See C.3.2(24/5). 14/3
- Any circumstances when the elaboration of a preelaborated package causes code to be executed. See C.4(12). 15/2
- Whether a partition can be restarted without reloading. See C.4(13). 16/2
- The effect of calling Current_Task from an entry body or interrupt handler. See C.7.1(19). 17/2

18/2 • For package Task_Attributes, limits on the number and size of task attributes, and how to configure any limits. See C.7.2(19).

19/2 • The metrics for the Task_Attributes package. See C.7.2(27).

20/2 • The details of the configuration used to generate the values of all metrics. See D(2).

21/2 • The maximum priority inversion a user task can experience from the implementation. See D.2.3(12/2).

22/2 • The amount of time that a task can be preempted for processing on behalf of lower-priority tasks. See D.2.3(13/2).

23/2 • The quantum values supported for round robin dispatching. See D.2.5(16/2).

24/2 • The accuracy of the detection of the exhaustion of the budget of a task for round robin dispatching. See D.2.5(17/2).

25/2 • Any conditions that cause the completion of the setting of the deadline of a task to be delayed for a multiprocessor. See D.2.6(32/2).

26/2 • Any conditions that cause the completion of the setting of the priority of a task to be delayed for a multiprocessor. See D.5.1(12.1/2).

27/2 • The metrics for Set_Priority. See D.5.1(14).

28/2 • The metrics for setting the priority of a protected object. See D.5.2(10).

29/2 • On a multiprocessor, any conditions that cause the completion of an aborted construct to be delayed later than what is specified for a single processor. See D.6(3).

30/2 • The metrics for aborts. See D.6(8).

31/2 • The values of Time_First, Time_Last, Time_Span_First, Time_Span_Last, Time_Span_Unit, and Tick for package Real_Time. See D.8(33).

32/2 • The properties of the underlying time base used in package Real_Time. See D.8(34).

33/2 • Any synchronization of package Real_Time with external time references. See D.8(35).

34/5 • Any aspects of the external environment that can interfere with package Real_Time. See D.8(36/5).

35/2 • The metrics for package Real_Time. See D.8(45).

36/2 • The minimum value of the delay expression of a delay_relative_statement that causes a task to actually be blocked. See D.9(7).

37/2 • The minimum difference between the value of the delay expression of a delay_until_statement and the value of Real_Time.Clock, that causes the task to actually be blocked. See D.9(8).

38/2 • The metrics for delay statements. See D.9(13).

39/2 • The upper bound on the duration of interrupt blocking caused by the implementation. See D.12(5).

40/2 • The metrics for entry-less protected objects. See D.12(12).

41/2 • The values of CPU_Time_First, CPU_Time_Last, CPU_Time_Unit, and CPU_Tick of package Execution_Time. See D.14(21/2).

42/3 • The properties of the mechanism used to implement package Execution_Time, including the values of the constants defined in the package. See D.14(22/2).

43/2 • The metrics for execution time. See D.14(27).

- The metrics for timing events. See D.15(24). 44/2
- The processor(s) on which the clock interrupt is handled; the processors on which each Interrupt_Id can be handled. See D.16.1(32). 44.1/3
- Whether the RPC-receiver is invoked from concurrent tasks, and if so, the number of such tasks. See E.5(25). 45/2
- Any techniques used to reduce cancellation errors in Numerics.Generic_Real_Arrays shall be documented. See G.3.1(86/2). 46/2
- Any techniques used to reduce cancellation errors in Numerics.Generic_Complex_Arrays shall be documented. See G.3.2(155/2). 47/2
- If a pragma Normalize_Scalars applies, the implicit initial values of scalar subtypes shall be documented. Such a value should be an invalid representation when possible; any cases when is it not shall be documented. See H.1(5/2). 48/2
- The range of effects for each bounded error and each unspecified effect. If the effects of a given erroneous construct are constrained, the constraints shall be documented. See H.2(1/5). 49/2
- For each inspection point, a mapping between each inspectable object and the machine resources where the object's value can be obtained shall be provided. See H.3.2(8). 50/2
- If a pragma Restrictions(No_Exceptions) is specified, the effects of all constructs where language-defined checks are still performed. See H.4(25). 51/2
- The interrupts to which a task entry may be attached. See J.7.1(12). 52/2
- The type of entry call invoked for an interrupt entry. See J.7.1(13). 53/2

M.2 Implementation-Defined Characteristics

The Ada language allows for certain machine dependences in a controlled manner. Each Ada implementation is required to document all implementation-defined characteristics: 1/5

- Whether or not each recommendation given in Implementation Advice is followed — see M.3 for a listing. See 1.1.2(37). 2/2
- Capacity limitations of the implementation. See 1.1.3(3). 3
- Variations from the standard that are impractical to avoid given the implementation's execution environment. See 1.1.3(6). 4
- Which code_statements cause external interactions. See 1.1.3(10). 5
- The coded representation for the text of an Ada program. See 2.1(4/6). 6
- The semantics of an Ada program whose text is not in Normalization Form C. See 2.1(4.1/6). 6.1/5
- *This paragraph was deleted.* 7/2
- The Unicode Character Database used to define character cateogories, case folding, and case mapping. See 2.1(17). 7.1/6
- The representation for an end of line. See 2.2(2/3). 8
- Maximum supported line length and lexical element length. See 2.2(14). 9
- Implementation-defined pragmas. See 2.8(14). 10
- Effect of pragma Optimize. See 2.8(27). 11

11.1/4 • The message string associated with the Assertion_Error exception raised by the failure of a predicate check if there is no applicable Predicate_Failure aspect. See 3.2.4(31).

13 • The predefined integer types declared in Standard. See 3.5.4(25).

14 • Any nonstandard integer types and the operators defined for them. See 3.5.4(26/5).

15 • Any nonstandard real types and the operators defined for them. See 3.5.6(8/5).

16 • What combinations of requested decimal precision and range are supported for floating point types. See 3.5.7(7).

17 • The predefined floating point types declared in Standard. See 3.5.7(16/5).

18 • The *small* of an ordinary fixed point type. See 3.5.9(8/2).

19 • What combinations of *small*, range, and *digits* are supported for fixed point types. See 3.5.9(10).

20/2 • The result of Tags.Wide_Wide_Expanded_Name for types declared within an unnamed block_statement. See 3.9(10).

20.1/2 • The sequence of characters of the value returned by Tags.Expanded_Name (respectively, Tags.Wide_Expanded_Name) when some of the graphic characters of Tags.Wide_Wide_Expanded_Name are not defined in Character (respectively, Wide_Character). See 3.9(10.1/2).

21 • Implementation-defined attributes. See 4.1.4(12/5).

21.1/5 • The value of the parameter to Empty for some container_aggregates. See 4.3.5(40).

21.2/5 • The maximum number of chunks for a parallel reduction expression without a chunk_specification. See 4.5.10(21).

21.3/2 • Rounding of real static expressions which are exactly half-way between two machine numbers. See 4.9(38/2).

21.4/5 • The maximum number of chunks for a parallel generalized iterator without a chunk_specification. See 5.5.2(10).

21.5/5 • The number of chunks for an array component iterator. See 5.5.2(11).

21.6/5 • Any extensions of the Global aspect. See 6.1.2(43).

21.7/5 • The circumstances in which the implementation passes in the null value for a view conversion of an access type used as an **out** parameter. See 6.4.1(19/5).

21.8/5 • Any extensions of the Default_Initial_Condition aspect. See 7.3.3(11).

22 • Any implementation-defined time types. See 9.6(6/3).

23 • The time base associated with relative delays. See 9.6(20).

24 • The time base of the type Calendar.Time. See 9.6(23).

25/2 • The time zone used for package Calendar operations. See 9.6(24/2).

26 • Any limit on delay_until_statements of select_statements. See 9.6(29).

26.1/2 • The result of Calendar.Formatting.Image if its argument represents more than 100 hours. See 9.6.1(86/5).

27/3 • *This paragraph was deleted.*

27.1/5 • Implementation-defined conflict check policies. See 9.10.1(5).

28 • The representation for a compilation. See 10.1(2).

- Any restrictions on compilations that contain multiple compilation_units. See 10.1(4). 29
- The mechanisms for creating an environment and for adding and replacing compilation units. See 10.1.4(3/2). 30
- The mechanisms for adding a compilation unit mentioned in a limited_with_clause to an environment. See 10.1.4(3/2). 30.1/2
- The manner of explicitly assigning library units to a partition. See 10.2(2). 31
- The implementation-defined means, if any, of specifying which compilation units are needed by a given compilation unit. See 10.2(2). 32
- The manner of designating the main subprogram of a partition. See 10.2(7). 33
- The order of elaboration of library_items. See 10.2(18). 34
- Parameter passing and function return for the main subprogram. See 10.2(21). 35
- The mechanisms for building and running partitions. See 10.2(24/5). 36
- The details of program execution, including program termination. See 10.2(25). 37
- The semantics of any nonactive partitions supported by the implementation. See 10.2(28/3). 38
- The information returned by Exception_Message. See 11.4.1(10.1/4). 39
- The result of Exceptions.Wide_Wide_Exception_Name for exceptions declared within an unnamed block_statement. See 11.4.1(12). 40/2
- The sequence of characters of the value returned by Exceptions.Exception_Name (respectively, Exceptions.Wide_Exception_Name) when some of the graphic characters of Exceptions.Wide_Wide_Exception_Name are not defined in Character (respectively, Wide_Character). See 11.4.1(12.1/2). 40.1/2
- The information returned by Exception_Information. See 11.4.1(13/2). 41
- Implementation-defined *policy*_identifiers and *assertion*_aspect_marks allowed in a pragma Assertion_Policy. See 11.4.2(9/5). 41.1/3
- The default assertion policy. See 11.4.2(10). 41.2/2
- Implementation-defined check names. See 11.5(27). 42
- Existence and meaning of second parameter of pragma Unsuppress. See 11.5(27.1/2). 42.1/2
- The cases that cause conflicts between the representation of the ancestors of a type_declaration. See 13.1(13.1/3). 42.2/2
- The interpretation of each representation aspect. See 13.1(20). 43/3
- Any restrictions placed upon the specification of representation aspects. See 13.1(20). 44/3
- Implementation-defined aspects, including the syntax for specifying such aspects and the legality rules for such aspects. See 13.1.1(38). 44.1/3
- The set of machine scalars. See 13.3(8.1/3). 44.2/2
- The meaning of Size for indefinite subtypes. See 13.3(48). 45
- The meaning of Object_Size for indefinite subtypes. See 13.3(58). 45.1/5
- The default external representation for a type tag. See 13.3(75/3). 46
- What determines whether a compilation unit is the same in two different partitions. See 13.3(76/6). 47

48 • Implementation-defined components. See 13.5.1(15).

49 • If Word_Size = Storage_Unit, the default bit ordering. See 13.5.3(5).

50/2 • The contents of the visible part of package System. See 13.7(2).

50.1/2 • The range of Storage_Elements.Storage_Offset, the modulus of Storage_Elements.Storage_-Element, and the declaration of Storage_Elements.Integer_Address. See 13.7.1(11).

51 • The contents of the visible part of package System.Machine_Code, and the meaning of code_statements. See 13.8(7).

51.1/2 • The result of unchecked conversion for instances with scalar result types whose result is not defined by the language. See 13.9(11).

52/2 • The effect of unchecked conversion for instances with nonscalar result types whose effect is not defined by the language. See 13.9(11).

53/2 • *This paragraph was deleted.*

54 • Whether or not the implementation provides user-accessible names for the standard pool type(s). See 13.11(17).

55/2 • The meaning of Storage_Size when neither the Storage_Size nor the Storage_Pool is specified for an access type. See 13.11(18).

56/2 • *This paragraph was deleted.*

56.1/4 • The effect of specifying aspect Default_Storage_Pool on an instance of a language-defined generic unit. See 13.11.3(5).

57/3 • *This paragraph was deleted.*

57.1/3 • Implementation-defined restrictions allowed in a pragma Restrictions. See 13.12(8.8/6).

58 • The consequences of violating limitations on Restrictions pragmas. See 13.12(9).

58.1/3 • Implementation-defined usage profiles allowed in a pragma Profile. See 13.12(15).

59/2 • The contents of the stream elements read and written by the Read and Write attributes of elementary types. See 13.13.2(9).

60 • The names and characteristics of the numeric subtypes declared in the visible part of package Standard. See A.1(3).

60.1/2 • The values returned by Strings.Hash. See A.4.9(3/2).

60.2/5 • The value returned by a call to a Text_Buffer Get procedure if any character in the returned sequence is not defined in Character. See A.4.12(34).

60.3/5 • The value returned by a call to a Text_Buffer Wide_Get procedure if any character in the returned sequence is not defined in Wide_Character. See A.4.12(34).

61 • The accuracy actually achieved by the elementary functions. See A.5.1(1).

62 • The sign of a zero result from some of the operators or functions in Numerics.Generic_Elementary_Functions, when Float_Type'Signed_Zeros is True. See A.5.1(46).

63 • The value of Numerics.Float_Random.Max_Image_Width. See A.5.2(27).

64 • The value of Numerics.Discrete_Random.Max_Image_Width. See A.5.2(27).

65/2 • *This paragraph was deleted.*

66 • The string representation of a random number generator's state. See A.5.2(38).

- *This paragraph was deleted.* 67/2
- The values of the Model_Mantissa, Model_Emin, Model_Epsilon, Model, Safe_First, and Safe_Last attributes, if the Numerics Annex is not supported. See A.5.3(72). 68
- *This paragraph was deleted.* 69/2
- The value of Buffer_Size in Storage_IO. See A.9(10). 70
- The external files associated with the standard input, standard output, and standard error files. See A.10(5). 71/2
- The accuracy of the value produced by Put. See A.10.9(36). 72
- Current size for a stream file for which positioning is not supported. See A.12.1(1.1/1). 72.1/1
- The meaning of Argument_Count, Argument, and Command_Name for package Command_Line. The bounds of type Command_Line.Exit_Status. See A.15(1). 73/2
- The interpretation of file names and directory names. See A.16(46/2). 73.1/2
- The maximum value for a file size in Directories. See A.16(87/2). 73.2/2
- The result for Directories.Size for a directory or special file. See A.16(93/2). 73.3/2
- The result for Directories.Modification_Time for a directory or special file. See A.16(95/2). 73.4/2
- The interpretation of a nonnull search pattern in Directories. See A.16(104/3). 73.5/2
- The results of a Directories search if the contents of the directory are altered while a search is in progress. See A.16(110/3). 73.6/2
- The definition and meaning of an environment variable. See A.17(1/2). 73.7/2
- The circumstances where an environment variable cannot be defined. See A.17(16/2). 73.8/2
- Environment names for which Set has the effect of Clear. See A.17(17/2). 73.9/2
- The value of Containers.Hash_Type'Modulus. The value of Containers.Count_Type'Last. See A.18.1(7/2). 73.10/2
- Implementation-defined convention names. See B.1(11/3). 74
- The meaning of link names. See B.1(36). 75
- The manner of choosing link names when neither the link name nor the address of an imported or exported entity is specified. See B.1(36). 76
- The effect of pragma Linker_Options. See B.1(37). 77
- The contents of the visible part of package Interfaces and its language-defined descendants. See B.2(1). 78
- Implementation-defined children of package Interfaces. See B.2(11). 79/2
- The definitions of certain types and constants in Interfaces.C. See B.3(41). 79.1/2
- The types Floating, Long_Floating, Binary, Long_Binary, Decimal_Element, and COBOL_Character; and the initializations of the variables Ada_To_COBOL and COBOL_To_Ada, in Interfaces.COBOL. See B.4(50). 80/1
- The types Fortran_Integer, Real, Double_Precision, and Character_Set in Interfaces.Fortran. See B.5(17). 80.1/1
- Implementation-defined intrinsic subprograms. See C.1(1/3). 81/2
- *This paragraph was deleted.* 82/2

83/2 • *This paragraph was deleted.*

83.1/3 • Any restrictions on a protected procedure or its containing type when an aspect Attach_handler or Interrupt_Handler is specified. See C.3.1(17).

83.2/3 • Any other forms of interrupt handler supported by the Attach_Handler and Interrupt_Handler aspects. See C.3.1(19).

84/2 • *This paragraph was deleted.*

85/4 • The semantics of some attributes and functions of an entity for which aspect Discard_Names is True. See C.5(7).

85.1/5 • The modulus and size of Test_and_Set_Flag. See C.6.3(8).

85.2/5 • The value used to represent the set value for Atomic_Test_and_Set. See C.6.3(10).

86 • The result of the Task_Identification.Image attribute. See C.7.1(7).

87/2 • The value of Current_Task when in a protected entry, interrupt handler, or finalization of a task attribute. See C.7.1(17/3).

88/2 • *This paragraph was deleted.*

88.1/1 • Granularity of locking for Task_Attributes. See C.7.2(16/1).

89/2 • *This paragraph was deleted.*

90/2 • *This paragraph was deleted.*

91 • The declarations of Any_Priority and Priority. See D.1(11).

92 • Implementation-defined execution resources. See D.1(15/5).

93 • Whether, on a multiprocessor, a task that is waiting for access to a protected object keeps its processor busy. See D.2.1(3).

94/2 • The effect of implementation-defined execution resources on task dispatching. See D.2.1(9/2).

95/2 • *This paragraph was deleted.*

96/2 • *This paragraph was deleted.*

97/2 • Implementation defined task dispatching policies. See D.2.2(18).

97.1/2 • The value of Default_Quantum in Dispatching.Round_Robin. See D.2.5(4).

98 • Implementation-defined *policy*_identifiers allowed in a pragma Locking_Policy. See D.3(4).

98.1/2 • The locking policy if no Locking_Policy pragma applies to any unit of a partition. See D.3(6).

99 • Default ceiling priorities. See D.3(10/4).

100 • The ceiling of any protected object used internally by the implementation. See D.3(16).

101 • Implementation-defined queuing policies. See D.4(1/5).

101.1/5 • Implementation-defined admission policies. See D.4.1(1).

102/2 • *This paragraph was deleted.*

103 • Any operations that implicitly require heap storage allocation. See D.7(8).

103.1/4 • When restriction No_Dynamic_CPU_Assignment applies to a partition, the processor on which a task with a CPU value of a Not_A_Specific_CPU will execute. See D.7(10).

103.2/2 • When restriction No_Task_Termination applies to a partition, what happens when a task terminates. See D.7(15.1/2).

- The behavior when restriction Max_Storage_At_Blocking is violated. See D.7(17/1). 103.3/2
- The behavior when restriction Max_Asynchronous_Select_Nesting is violated. See D.7(18/1). 103.4/2
- The behavior when restriction Max_Tasks is violated. See D.7(19). 103.5/2
- Whether the use of pragma Restrictions results in a reduction in program code or data size or execution time. See D.7(20). 104/2
- *This paragraph was deleted.* 105/2
- *This paragraph was deleted.* 106/2
- The value of Barrier_Limit'Last in Synchronous_Barriers. See D.10.1(4/3). 106.1/3
- When an aborted task that is waiting on a Synchronous_Barrier is aborted. See D.10.1(13/3). 106.2/3
- *This paragraph was deleted.* 107/2
- The value of Min_Handler_Ceiling in Execution_Time.Group_Budgets. See D.14.2(7/2). 107.1/3
- The value of CPU_Range'Last in System.Multiprocessors. See D.16(4/3). 107.2/3
- The processor on which the environment task executes in the absence of a value for the aspect CPU. See D.16(13/3). 107.3/3
- The means for creating and executing distributed programs. See E(5). 108
- Any events that can result in a partition becoming inaccessible. See E.1(7). 109
- The scheduling policies, treatment of priorities, and management of shared resources between partitions in certain cases. See E.1(11). 110
- *This paragraph was deleted.* 111/1
- Whether the execution of the remote subprogram is immediately aborted as a result of cancellation. See E.4(13). 112
- The range of type System.RPC.Partition_Id. See E.5(14). 112.1/2
- *This paragraph was deleted.* 113/2
- Implementation-defined interfaces in the PCS. See E.5(26). 114
- The values of named numbers in the package Decimal. See F.2(7). 115
- The value of Max_Picture_Length in the package Text_IO.Editing See F.3.3(16). 116
- The value of Max_Picture_Length in the package Wide_Text_IO.Editing See F.3.4(5). 117
- The value of Max_Picture_Length in the package Wide_Wide_Text_IO.Editing See F.3.5(5). 117.1/2
- The accuracy actually achieved by the complex elementary functions and by other complex arithmetic operations. See G.1(1). 118
- The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Types, when Real'Signed_Zeros is True. See G.1.1(53). 119
- The sign of a zero result (or a component thereof) from any operator or function in Numerics.Generic_Complex_Elementary_Functions, when Complex_Types.Real'Signed_Zeros is True. See G.1.2(45). 120
- Whether the strict mode or the relaxed mode is the default. See G.2(2). 121
- The result interval in certain cases of fixed-to-float conversion. See G.2.1(10). 122
- The result of a floating point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.1(13). 123

124 • The result interval for division (or exponentiation by a negative exponent), when the floating point hardware implements division as multiplication by a reciprocal. See G.2.1(16).

125 • The definition of *close result set*, which determines the accuracy of certain fixed point multiplications and divisions. See G.2.3(5).

126 • Conditions on a *universal_real* operand of a fixed point multiplication or division for which the result shall be in the *perfect result set*. See G.2.3(22).

127 • The result of a fixed point arithmetic operation in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.3(27).

128 • The result of an elementary function reference in overflow situations, when the Machine_Overflows attribute of the result type is False. See G.2.4(4).

129 • The value of the *angle threshold*, within which certain elementary functions, complex arithmetic operations, and complex elementary functions yield results conforming to a maximum relative error bound. See G.2.4(10).

130 • The accuracy of certain elementary functions for parameters beyond the angle threshold. See G.2.4(10).

131 • The result of a complex arithmetic operation or complex elementary function reference in overflow situations, when the Machine_Overflows attribute of the corresponding real type is False. See G.2.6(5).

132 • The accuracy of certain complex arithmetic operations and certain complex elementary functions for parameters (or components thereof) beyond the angle threshold. See G.2.6(8).

132.1/2 • The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Real_Matrix. See G.3.1(81/2).

132.2/2 • The accuracy requirements for the subprograms Solve, Inverse, Determinant, Eigenvalues and Eigensystem for type Complex_Matrix. See G.3.2(149/2).

133/2 • *This paragraph was deleted.*

134/2 • *This paragraph was deleted.*

134.1/5 • The consequences of violating No_Hidden_Indirect_Globals. See H.4(23.9/5).

135/2 • *This paragraph was deleted.*

136/2 • *This paragraph was deleted.*

136.1/2 • Implementation-defined *policy*_identifiers allowed in a pragma Partition_Elaboration_Policy. See H.6(4/2).

M.3 Implementation Advice

1/5 This Reference Manual sometimes gives advice about handling certain target machine dependences. Each Ada implementation is required to document whether that advice is followed:

2/2 • Program_Error should be raised when an unsupported Specialized Needs Annex feature is used at run time. See 1.1.3(20).

3/2 • Implementation-defined extensions to the functionality of a language-defined library unit should be provided by adding children to the library unit. See 1.1.3(21).

4/2 • If a bounded error or erroneous execution is detected, Program_Error should be raised. See 1.1.5(12).

- Implementation-defined pragmas should have no semantic effect for error-free programs. See 2.8(16/3). 5/2
- Implementation-defined pragmas should not make an illegal program legal, unless they complete a declaration or configure the library_items in an environment. See 2.8(19). 6/2
- Long_Integer should be declared in Standard if the target supports 32-bit arithmetic. No other named integer subtypes should be declared in Standard. See 3.5.4(28). 7/2
- For a two's complement target, modular types with a binary modulus up to System.Max_Int*2+2 should be supported. A nonbinary modulus up to Integer'Last should be supported. See 3.5.4(29). 8/2
- Program_Error should be raised for the evaluation of S'Pos for an enumeration type, if the value of the operand does not correspond to the internal code for any enumeration literal of the type. See 3.5.5(8). 9/2
- Long_Float should be declared in Standard if the target supports 11 or more digits of precision. No other named float subtypes should be declared in Standard. See 3.5.7(17). 10/2
- Multidimensional arrays should be represented in row-major order, unless the array has convention Fortran. See 3.6.2(11/3). 11/2
- Tags.Internal_Tag should return the tag of a type, if one exists, whose innermost master is a master of the point of evaluation of the function call. See 3.9(26.1/6). 12/6
- A real static expression with a nonformal type that is not part of a larger static expression should be rounded the same as the target system. See 4.9(38.1/2). 13/2
- For each language-defined private type T, T'Image should generate an image that would be meaningful based only on the relevant public interfaces. See 4.10(56). 13.1/5
- The value of Duration'Small should be no greater than 100 microseconds. See 9.6(30). 14/2
- The time base for delay_relative_statements should be monotonic. See 9.6(31/5). 15/2
- Leap seconds should be supported if the target system supports them. Otherwise, operations in Calendar.Formatting should return results consistent with no leap seconds. See 9.6.1(89/2). 16/2
- *This paragraph was deleted.* 17/5
- A type declared in a preelaborated package should have the same representation in every elaboration of a given version of the package. See 10.2.1(12). 18/2
- Exception_Information should provide information useful for debugging, and should include the Exception_Name and Exception_Message. See 11.4.1(19). 19/2
- Exception_Message by default should be short, provide information useful for debugging, and should not include the Exception_Name. See 11.4.1(19). 20/2
- Code executed for checks that have been suppressed should be minimized. See 11.5(28). 21/2
- The recommended level of support for all representation items should be followed. See 13.1(28/5). 22/2
- Storage allocated to objects of a packed type should be minimized. See 13.2(6). 23/2
- The recommended level of support for the Pack aspect should be followed. See 13.2(9). 24/3
- For an array X, X'Address should point at the first component of the array rather than the array bounds. See 13.3(14). 25/2
- The recommended level of support for the Address attribute should be followed. See 13.3(19). 26/2
- For any tagged specific subtype *S*, *S*'Class'Alignment should equal *S*'Alignment. See 13.3(28). 26.1/3

27/2 • The recommended level of support for the Alignment attribute should be followed. See 13.3(35).

28/2 • The Size of an array object should not include its bounds. See 13.3(41.1/2).

29/5 • If the Size of a subtype is nonconfirming and allows for efficient independent addressability, then the Object_Size of the subtype (unless otherwise specified) should equal the Size of the subtype. See 13.3(52).

30/2 • A Size clause on a composite subtype should not affect the internal layout of components. See 13.3(53).

31/2 • The recommended level of support for the Size attribute should be followed. See 13.3(56).

31.1/5 • An Object_Size clause on a composite type should not affect the internal layout of components. See 13.3(58).

31.2/5 • If S is a definite first subtype for which Object_Size is not specified, S'Object_Size should be the smallest multiple of the storage element size larger than or equal to S'Size that is consistent with the alignment of S. See 13.3(58).

31.3/5 • The recommended level of support for the Object_Size attribute should be followed. See 13.3(58).

31.4/5 • The Size of most objects of a subtype should equal the Object_Size of the subtype. See 13.3(58).

32/2 • The recommended level of support for the Component_Size attribute should be followed. See 13.3(73).

33/2 • The recommended level of support for enumeration_representation_clauses should be followed. See 13.4(10).

34/2 • The recommended level of support for record_representation_clauses should be followed. See 13.5.1(22).

35/2 • If a component is represented using a pointer to the actual data of the component which is contiguous with the rest of the object, then the storage place attributes should reflect the place of the actual data. If a component is allocated discontiguously from the rest of the object, then a warning should be generated upon reference to one of its storage place attributes. See 13.5.2(5).

36/2 • The recommended level of support for the nondefault bit ordering should be followed. See 13.5.3(8).

37/2 • Type System.Address should be a private type. See 13.7(37).

38/2 • Operations in System and its children should reflect the target environment; operations that do not make sense should raise Program_Error. See 13.7.1(16).

39/2 • Since the Size of an array object generally does not include its bounds, the bounds should not be part of the converted data in an instance of Unchecked_Conversion. See 13.9(14/2).

40/2 • There should not be unnecessary runtime checks on the result of an Unchecked_Conversion; the result should be returned by reference when possible. Restrictions on Unchecked_Conversions should be avoided. See 13.9(15).

41/2 • The recommended level of support for Unchecked_Conversion should be followed. See 13.9(17).

42/2 • Any cases in which heap storage is dynamically allocated other than as part of the evaluation of an allocator should be documented. See 13.11(23).

43/2 • A default storage pool for an access-to-constant type should not have overhead to support deallocation of individual objects. See 13.11(24).

- Usually, a storage pool for an access discriminant or access parameter should be created at the point of an allocator, and be reclaimed when the designated object becomes inaccessible. For other anonymous access types, the pool should be created at the point where the type is elaborated and may have no mechanism for the deallocation of individual objects. See 13.11(25). 44/2
- For a standard storage pool, an instance of Unchecked_Deallocation should actually reclaim the storage. See 13.11.2(17). 45/2
- A call on an instance of Unchecked_Deallocation with a nonnull access value should raise Program_Error if the actual access type of the instance is a type for which the Storage_Size has been specified to be zero or is defined by the language to be zero. See 13.11.2(17.1/3). 45.1/3
- Streams.Storage.Bounded.Stream_Type objects should be implemented without implicit pointers or dynamic allocation. See 13.13.1(37). 45.2/5
- If not specified, the value of Stream_Size for an elementary type should be the number of bits that corresponds to the minimum number of stream elements required by the first subtype of the type, rounded up to the nearest factor or multiple of the word size that is also a multiple of the stream element size. See 13.13.2(1.6/2). 46/2
- The recommended level of support for the Stream_Size attribute should be followed. See 13.13.2(1.8/2). 47/2
- If an implementation provides additional named predefined integer types, then the names should end with “Integer”. If an implementation provides additional named predefined floating point types, then the names should end with “Float”. See A.1(52). 48/2
- Implementation-defined operations on Wide_Character, Wide_String, Wide_Wide_Character, and Wide_Wide_String should be child units of Wide_Characters or Wide_Wide_Characters. See A.3.1(7/3). 49/2
- The string returned by Wide_Characters.Handling.Character_Set_Version should include either “10646:” or “Unicode”. See A.3.5(62). 49.1/3
- Bounded string objects should not be implemented by implicit pointers and dynamic allocation. See A.4.4(106). 50/2
- Strings.Hash should be good a hash function, returning a wide spread of values for different string values, and similar strings should rarely return the same value. See A.4.9(12/2). 51/2
- If an implementation supports other string encoding schemes, a child of Ada.Strings similar to UTF_Encoding should be defined. See A.4.11(107/3). 51.1/3
- Bounded buffer objects should be implemented without dynamic allocation. See A.4.12(36). 51.2/5
- Any storage associated with an object of type Generator of the random number packages should be reclaimed on exit from the scope of the object. See A.5.2(46). 52/2
- Each value of Initiator passed to Reset for the random number packages should initiate a distinct sequence of random numbers, or, if that is not possible, be at least a rapidly varying function of the initiator value. See A.5.2(47). 53/2
- Get_Immediate should be implemented with unbuffered input; input should be available immediately; line-editing should be disabled. See A.10.7(23). 54/2
- Package Directories.Information should be provided to retrieve other information about a file. See A.16(124/2). 55/2
- Directories.Start_Search and Directories.Search should raise Name_Error for malformed patterns. See A.16(125). 56/3

57/2 • Directories.Rename should be supported at least when both New_Name and Old_Name are simple names and New_Name does not identify an existing external file. See A.16(126/2).

57.1/3 • Directories.Hierarchical_File_Names should be provided for systems with hierarchical file naming, and should not be provided on other systems. See A.16.1(36/3).

58/2 • If the execution environment supports subprocesses, the current environment variables should be used to initialize the environment variables of a subprocess. See A.17(32/2).

59/2 • Changes to the environment variables made outside the control of Environment_Variables should be reflected immediately. See A.17(33/2).

60/2 • Containers.Hash_Type'Modulus should be at least 2**32. Containers.Count_Type'Last should be at least 2**31–1. See A.18.1(8/2).

61/2 • The worst-case time complexity of Element for Containers.Vector should be $O(\log N)$. See A.18.2(256/2).

62/2 • The worst-case time complexity of Append with Count = 1 when N is less than the capacity for Containers.Vector should be $O(\log N)$. See A.18.2(257/2).

63/2 • The worst-case time complexity of Prepend with Count = 1 and Delete_First with Count=1 for Containers.Vectors should be $O(N \log N)$. See A.18.2(258/2).

64/2 • The worst-case time complexity of a call on procedure Sort of an instance of Containers.Vectors.Generic_Sorting should be $O(N^{**}2)$, and the average time complexity should be better than $O(N^{**}2)$. See A.18.2(259/2).

65/2 • Containers.Vectors.Generic_Sorting.Sort and Containers.Vectors.Generic_Sorting.Merge should minimize copying of elements. See A.18.2(260/2).

66/2 • Containers.Vectors.Move should not copy elements, and should minimize copying of internal data structures. See A.18.2(261/2).

67/2 • If an exception is propagated from a vector operation, no storage should be lost, nor any elements removed from a vector unless specified by the operation. See A.18.2(262/2).

68/2 • The worst-case time complexity of Element, Insert with Count=1, and Delete with Count=1 for Containers.Doubly_Linked_Lists should be $O(\log N)$. See A.18.3(160/2).

69/2 • A call on procedure Sort of an instance of Containers.Doubly_Linked_Lists.Generic_Sorting should have an average time complexity better than $O(N^{**}2)$ and worst case no worse than $O(N^{**}2)$. See A.18.3(161/2).

70/2 • Containers.Doubly_Linked_Lists.Move should not copy elements, and should minimize copying of internal data structures. See A.18.3(162/2).

71/2 • If an exception is propagated from a list operation, no storage should be lost, nor any elements removed from a list unless specified by the operation. See A.18.3(163/2).

72/2 • Move for a map should not copy elements, and should minimize copying of internal data structures. See A.18.4(83/2).

73/2 • If an exception is propagated from a map operation, no storage should be lost, nor any elements removed from a map unless specified by the operation. See A.18.4(84/2).

74/2 • The average time complexity of Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter for Containers.Hashed_Maps should be $O(\log N)$. The average time complexity of the subprograms of Containers.Hashed_Maps that take a cursor parameter should be $O(1)$. The average time complexity of Containers.Hashed_Maps.Reserve_Capacity should be $O(N)$. See A.18.5(62/2).

- The worst-case time complexity of Element, Insert, Include, Replace, Delete, Exclude, and Find operations that take a key parameter for Containers.Ordered_Maps should be *O*((log *N*)**2) or better. The worst-case time complexity of the subprograms of Containers.Ordered_Maps that take a cursor parameter should be *O*(1). See A.18.6(95/2). 75/2
- Move for sets should not copy elements, and should minimize copying of internal data structures. See A.18.7(104/2). 76/2
- If an exception is propagated from a set operation, no storage should be lost, nor any elements removed from a set unless specified by the operation. See A.18.7(105/2). 77/2
- The average time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations of Containers.Hashed_Sets that take an element parameter should be *O*(log *N*). The average time complexity of the subprograms of Containers.Hashed_Sets that take a cursor parameter should be *O*(1). The average time complexity of Containers.Hashed_Sets.-Reserve_Capacity should be *O*(*N*). See A.18.8(88/2). 78/2
- The worst-case time complexity of the Insert, Include, Replace, Delete, Exclude, and Find operations of Containers.Ordered_Sets that take an element parameter should be *O*((log *N*)**2). The worst-case time complexity of the subprograms of Containers.Ordered_Sets that take a cursor parameter should be *O*(1). See A.18.9(116/2). 79/2
- The worst-case time complexity of the Element, Parent, First_Child, Last_Child, Next_Sibling, Previous_Sibling, Insert_Child with Count=1, and Delete operations of Containers.Multiway_Trees should be *O*(log *N*). See A.18.10(231/3). 79.1/3
- Containers.Multiway_Trees.Move should not copy elements, and should minimize copying of internal data structures. See A.18.10(232/3). 79.2/3
- If an exception is propagated from a tree operation, no storage should be lost, nor any elements removed from a tree unless specified by the operation. See A.18.10(233/3). 79.3/3
- Move and Swap in Containers.Indefinite_Holders should not copy any elements, and should minimize copying of internal data structures. See A.18.18(73/5). 79.4/5
- If an exception is propagated from a holder operation, no storage should be lost, nor should the element be removed from a holder container unless specified by the operation. See A.18.18(74/3). 79.5/3
- Bounded vector objects should be implemented without implicit pointers or dynamic allocation. See A.18.19(16/3). 79.6/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded vectors. See A.18.19(17/3). 79.7/3
- Bounded list objects should be implemented without implicit pointers or dynamic allocation. See A.18.20(19/3). 79.8/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded lists. See A.18.20(20/3). 79.9/3
- Bounded hashed map objects should be implemented without implicit pointers or dynamic allocation. See A.18.21(21/3). 79.10/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded hashed maps. See A.18.21(22/3). 79.11/3
- Bounded ordered map objects should be implemented without implicit pointers or dynamic allocation. See A.18.22(18/3). 79.12/3
- The implementation advice for procedure Move to minimize copying does not apply to bounded ordered maps. See A.18.22(19/3). 79.13/3

79.14/3 • Bounded hashed set objects should be implemented without implicit pointers or dynamic allocation. See A.18.23(20/3).

79.15/3 • The implementation advice for procedure Move to minimize copying does not apply to bounded hashed sets. See A.18.23(21/3).

79.16/3 • Bounded ordered set objects should be implemented without implicit pointers or dynamic allocation. See A.18.24(17/3).

79.17/3 • The implementation advice for procedure Move to minimize copying does not apply to bounded ordered sets. See A.18.24(18/3).

79.18/3 • Bounded tree objects should be implemented without implicit pointers or dynamic allocation. See A.18.25(19/3).

79.19/3 • The implementation advice for procedure Move to minimize copying does not apply to bounded trees. See A.18.25(20/3).

80/2 • Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). See A.18.26(10/2).

81/2 • Containers.Generic_Array_Sort and Containers.Generic_Constrained_Array_Sort should minimize copying of elements. See A.18.26(11/2).

81.1/3 • Containers.Generic_Sort should have an average time complexity better than *O*(*N***2) and worst case no worse than *O*(*N***2). See A.18.26(12/3).

81.2/3 • Containers.Generic_Sort should minimize calls to the generic formal Swap. See A.18.26(13/3).

81.3/3 • Bounded queue objects should be implemented without implicit pointers or dynamic allocation. See A.18.29(13/3).

81.4/3 • Bounded priority queue objects should be implemented without implicit pointers or dynamic allocation. See A.18.31(14/3).

81.5/5 • Bounded holder objects should be implemented without dynamic allocation. See A.18.32(15/5).

82/6 • If Export is supported for a language, the main subprogram should be able to be written in that language. Subprograms named "adainit" and "adafinal" should be provided for elaboration and finalization of the environment task. See B.1(39/3).

83/3 • Automatic elaboration of preelaborated packages should be provided when specifying the Export aspect as True is supported. See B.1(40/3).

84/3 • For each supported convention *L* other than Intrinsic, specifying the aspects Import and Export should be supported for objects of *L*-compatible types and for subprograms, and aspect Convention should be supported for *L*-eligible types and for subprograms. See B.1(41/5).

85/2 • If an interface to C, COBOL, or Fortran is provided, the corresponding package or packages described in Annex B, “Interface to Other Languages” should also be provided. See B.2(13/3).

86/2 • The constants nul, wide_nul, char16_nul, and char32_nul in package Interfaces.C should have a representation of zero. See B.3(62.5/3).

87/2 • If C interfacing is supported, the interface correspondences between Ada and C should be supported. See B.3(71).

87.1/5 • If the C implementation supports unsigned long long and long long, unsigned_long_long and long_long should be supported. See B.3(71).

88/2 • If COBOL interfacing is supported, the interface correspondences between Ada and COBOL should be supported. See B.4(98).

- If Fortran interfacing is supported, the interface correspondences between Ada and Fortran should be supported. See B.5(26). 89/2
- The machine code or intrinsics support should allow access to all operations normally available to assembly language programmers for the target environment. See C.1(3). 90/2
- Interface to assembler should be supported; the default assembler should be associated with the convention identifier Assembler. See C.1(4/3). 91/2
- If an entity is exported to assembly language, then the implementation should allocate it at an addressable location even if not otherwise referenced from the Ada code. A call to a machine code or assembler subprogram should be treated as if it can read or update every object that is specified as exported. See C.1(5). 92/5
- Little or no overhead should be associated with calling intrinsic and machine-code subprograms. See C.1(10). 93/2
- Intrinsic subprograms should be provided to access any machine operations that provide special capabilities or efficiency not normally available. See C.1(16). 94/2
- If the Ceiling_Locking policy is not in effect and the target system allows for finer-grained control of interrupt blocking, a means for the application to specify which interrupts are to be blocked during protected actions should be provided. See C.3(28/2). 95/2
- Interrupt handlers should be called directly by the hardware. See C.3.1(20). 96/2
- Violations of any implementation-defined restrictions on interrupt handlers should be detected before run time. See C.3.1(21). 97/2
- If implementation-defined forms of interrupt handler procedures are supported, then for each such form of a handler, a type analogous to Parameterless_Handler should be specified in a child package of Interrupts, with the same operations as in the predefined package Interrupts. See C.3.2(25). 98/2
- Preelaborated packages should be implemented such that little or no code is executed at run time for the elaboration of entities. See C.4(14). 99/2
- If aspect Discard_Names is True for an entity, then the amount of storage used for storing names associated with that entity should be reduced. See C.5(8/4). 100/4
- A load or store of a volatile object whose size is a multiple of System.Storage_Unit and whose alignment is nonzero, should be implemented by accessing exactly the bits of the object and no others. See C.6(22/5). 101/2
- A load or store of an atomic object should be implemented by a single load or store instruction. See C.6(23/2). 102/2
- If the target domain requires deterministic memory use at run time, storage for task attributes should be pre-allocated statically and the number of attributes pre-allocated should be documented. See C.7.2(30). 103/2
- Finalization of task attributes and reclamation of associated storage should be performed as soon as possible after task termination. See C.7.2(30.1/2). 104/2
- Names that end with "_Locking" should be used for implementation-defined locking policies. See D.3(17). 105/2
- Names that end with "_Queuing" should be used for implementation-defined queuing policies. See D.4(16). 106/2
- The abort_statement should not require the task executing the statement to block. See D.6(9). 107/2

108/2 • On a multi-processor, the delay associated with aborting a task on another processor should be bounded. See D.6(10).

109/2 • When feasible, specified restrictions should be used to produce a more efficient implementation. See D.7(21).

110/2 • When appropriate, mechanisms to change the value of Tick should be provided. See D.8(47).

111/2 • Calendar.Clock and Real_Time.Clock should be transformations of the same time base. See D.8(48).

112/2 • The “best” time base which exists in the underlying system should be available to the application through Real_Time.Clock. See D.8(49).

112.1/3 • On a multiprocessor system, each processor should have a separate and disjoint ready queue. See D.13(9).

113/2 • When appropriate, implementations should provide configuration mechanisms to change the value of Execution_Time.CPU_Tick. See D.14(29/2).

114/2 • For a timing event, the handler should be executed directly by the real-time clock interrupt mechanism. See D.15(25).

114.1/5 • Starting a protected action on a protected object statically assigned to a processor should not use busy-waiting. See D.16(16).

114.2/3 • Each dispatching domain should have separate and disjoint ready queues. See D.16.1(31).

115/2 • The PCS should allow for multiple tasks to call the RPC-receiver. See E.5(28).

116/2 • The System.RPC.Write operation should raise Storage_Error if it runs out of space when writing an item. See E.5(29).

117/2 • If COBOL (respectively, C) is supported in the target environment, then interfacing to COBOL (respectively, C) should be supported as specified in Annex B. See F(7/3).

118/2 • Packed decimal should be used as the internal representation for objects of subtype *S* when *S*'Machine_Radix = 10. See F.1(2).

119/2 • If Fortran (respectively, C) is supported in the target environment, then interfacing to Fortran (respectively, C) should be supported as specified in Annex B. See G(7/3).

120/2 • Mixed real and complex operations (as well as pure-imaginary and complex operations) should not be performed by converting the real (resp. pure-imaginary) operand to complex. See G.1.1(56/5).

121/3 • If Real'Signed_Zeros is True for Numerics.Generic_Complex_Types, a rational treatment of the signs of zero results and result components should be provided. See G.1.1(58).

122/3 • If Complex_Types.Real'Signed_Zeros is True for Numerics.Generic_Complex_Elementary_Functions, a rational treatment of the signs of zero results and result components should be provided. See G.1.2(49).

123/2 • For elementary functions, the forward trigonometric functions without a Cycle parameter should not be implemented by calling the corresponding version with a Cycle parameter. Log without a Base parameter should not be implemented by calling Log with a Base parameter. See G.2.4(19).

124/2 • For complex arithmetic, the Compose_From_Polar function without a Cycle parameter should not be implemented by calling Compose_From_Polar with a Cycle parameter. See G.2.6(15).

- Solve and Inverse for Numerics.Generic_Real_Arrays should be implemented using established techniques such as LU decomposition and the result should be refined by an iteration on the residuals. See G.3.1(88/3). 125/2
- The equality operator should be used to test that a matrix in Numerics.Generic_Real_Arrays is symmetric. See G.3.1(90/2). 126/2
- An implementation should minimize the circumstances under which the algorithm used for Numerics.Generic_Real_Arrays.Eigenvalues and Numerics.Generic_Real_Arrays.Eigensystem fails to converge. See G.3.1(91/3). 126.1/3
- Solve and Inverse for Numerics.Generic_Complex_Arrays should be implemented using established techniques and the result should be refined by an iteration on the residuals. See G.3.2(158/3). 127/2
- The equality and negation operators should be used to test that a matrix is Hermitian. See G.3.2(160/2). 128/2
- An implementation should minimize the circumstances under which the algorithm used for Numerics.Generic_Complex_Arrays.Eigenvalues and Numerics.Generic_Complex_Arrays.Eigensystem fails to converge. See G.3.2(160.1/3). 128.1/3
- Mixed real and complex operations should not be performed by converting the real operand to complex. See G.3.2(161/2). 129/2
- The information produced by pragma Reviewable should be provided in both a human-readable and machine-readable form, and the latter form should be documented. See H.3.1(19). 130/2
- Object code listings should be provided both in a symbolic format and in a numeric format. See H.3.1(20). 131/2
- If the partition elaboration policy is Sequential and the Environment task becomes permanently blocked during elaboration, then the partition should be immediately terminated. See H.6(15/3). 132/3
- When applied to a generic unit, a program unit pragma that is not a library unit pragma should apply to each instance of the generic unit for which there is not an overriding pragma applied directly to the instance. See J.15(9/5). 132.1/5

Annex N
(informative)
Glossary

This paragraph was deleted. 1/5

Informal definitions of various terms are now found in 1.3.

Annex P
(informative)
Syntax Summary

This Annex summarizes the complete syntax of the language.

P.1 Syntax Rules

This subclause lists the complete syntax of the language in the order it appears in this Reference Manual. See 1.1.4 for a description of the notation used. 1/5

2.3:
```
identifier ::=
  identifier_start {identifier_start | identifier_extend}
```

2.3:
```
identifier_start ::=
    letter_uppercase
  | letter_lowercase
  | letter_titlecase
  | letter_modifier
  | letter_other
  | number_letter
```

2.3:
```
identifier_extend ::=
    mark_non_spacing
  | mark_spacing_combining
  | number_decimal
  | punctuation_connector
```

2.4:
```
numeric_literal ::= decimal_literal | based_literal
```

2.4.1:
```
decimal_literal ::= numeral [.numeral] [exponent]
```

2.4.1:
```
numeral ::= digit {[underline] digit}
```

2.4.1:
```
exponent ::= E [+] numeral | E – numeral
```

2.4.1:
```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2.4.2:
```
based_literal ::=
  base # based_numeral [.based_numeral] # [exponent]
```

2.4.2:
```
base ::= numeral
```

2.4.2:
```
based_numeral ::=
  extended_digit {[underline] extended_digit}
```

2.4.2:
```
extended_digit ::= digit | A | B | C | D | E | F
```

2.5:
character_literal ::= 'graphic_character'

2.6:
string_literal ::= "{string_element}"

2.6:
string_element ::= "" | *non_quotation_mark*_graphic_character

2.7:
comment ::= --{*non_end_of_line*_character}

2.8:
pragma ::=
  pragma identifier [(pragma_argument_association {, pragma_argument_association})];

2.8:
pragma_argument_association ::=
    [*pragma_argument*_identifier =>] name
  | [*pragma_argument*_identifier =>] expression
  | *pragma_argument*_aspect_mark => name
  | *pragma_argument*_aspect_mark => expression

3.1:
basic_declaration ::=
    type_declaration | subtype_declaration
  | object_declaration | number_declaration
  | subprogram_declaration | abstract_subprogram_declaration
  | null_procedure_declaration | expression_function_declaration
  | package_declaration | renaming_declaration
  | exception_declaration | generic_declaration
  | generic_instantiation

3.1:
defining_identifier ::= identifier

3.2.1:
type_declaration ::= full_type_declaration
  | incomplete_type_declaration
  | private_type_declaration
  | private_extension_declaration

3.2.1:
full_type_declaration ::=
    type defining_identifier [known_discriminant_part] **is** type_definition
      [aspect_specification];
  | task_type_declaration
  | protected_type_declaration

3.2.1:
type_definition ::=
    enumeration_type_definition | integer_type_definition
  | real_type_definition | array_type_definition
  | record_type_definition | access_type_definition
  | derived_type_definition | interface_type_definition

3.2.2:
subtype_declaration ::=
  subtype defining_identifier **is** subtype_indication
      [aspect_specification];

3.2.2:
subtype_indication ::= [null_exclusion] subtype_mark [constraint]

3.2.2:
subtype_mark ::= *subtype*_name

3.2.2:
constraint ::= scalar_constraint | composite_constraint

3.2.2:
scalar_constraint ::=
    range_constraint | digits_constraint | delta_constraint

3.2.2:
composite_constraint ::=
    index_constraint | discriminant_constraint

3.3.1:
object_declaration ::=
   defining_identifier_list : [**aliased**] [**constant**] subtype_indication [:= expression]
      [aspect_specification];
 | defining_identifier_list : [**aliased**] [**constant**] access_definition [:= expression]
      [aspect_specification];
 | defining_identifier_list : [**aliased**] [**constant**] array_type_definition [:= expression]
      [aspect_specification];
 | single_task_declaration
 | single_protected_declaration

3.3.1:
defining_identifier_list ::=
  defining_identifier {, defining_identifier}

3.3.2:
number_declaration ::=
    defining_identifier_list : **constant** := *static_*expression;

3.4:
derived_type_definition ::=
   [**abstract**] [**limited**] **new** *parent_*subtype_indication [[**and** interface_list] record_extension_part]

3.5:
range_constraint ::= **range** range

3.5:
range ::= range_attribute_reference
  | simple_expression .. simple_expression

3.5.1:
enumeration_type_definition ::=
  (enumeration_literal_specification {, enumeration_literal_specification})

3.5.1:
enumeration_literal_specification ::=
   defining_identifier | defining_character_literal

3.5.1:
defining_character_literal ::= character_literal

3.5.4:
integer_type_definition ::=
   signed_integer_type_definition | modular_type_definition

3.5.4:
signed_integer_type_definition ::=
   range *static_*simple_expression .. *static_*simple_expression

3.5.4:
modular_type_definition ::= **mod** *static_*expression

3.5.6:
real_type_definition ::=
   floating_point_definition | fixed_point_definition

3.5.7:
floating_point_definition ::=
  digits *static*_expression [real_range_specification]

3.5.7:
real_range_specification ::=
  range *static*_simple_expression .. *static*_simple_expression

3.5.9:
fixed_point_definition ::=
  ordinary_fixed_point_definition | decimal_fixed_point_definition

3.5.9:
ordinary_fixed_point_definition ::=
  delta *static*_expression real_range_specification

3.5.9:
decimal_fixed_point_definition ::=
  delta *static*_expression **digits** *static*_expression [real_range_specification]

3.5.9:
digits_constraint ::=
  digits *static*_simple_expression [range_constraint]

3.6:
array_type_definition ::=
  unconstrained_array_definition | constrained_array_definition

3.6:
unconstrained_array_definition ::=
  array(index_subtype_definition {, index_subtype_definition}) **of** component_definition

3.6:
index_subtype_definition ::= subtype_mark **range** <>

3.6:
constrained_array_definition ::=
  array (discrete_subtype_definition {, discrete_subtype_definition}) **of** component_definition

3.6:
discrete_subtype_definition ::= *discrete*_subtype_indication | range

3.6:
component_definition ::=
  [**aliased**] subtype_indication
| [**aliased**] access_definition

3.6.1:
index_constraint ::= (discrete_range {, discrete_range})

3.6.1:
discrete_range ::= *discrete*_subtype_indication | range

3.7:
discriminant_part ::= unknown_discriminant_part | known_discriminant_part

3.7:
unknown_discriminant_part ::= (<>)

3.7:
known_discriminant_part ::=
  (discriminant_specification {; discriminant_specification})

3.7:
discriminant_specification ::=
  defining_identifier_list : [null_exclusion] subtype_mark [:= default_expression]
    [aspect_specification]
 | defining_identifier_list : access_definition [:= default_expression]
    [aspect_specification]

3.7:
default_expression ::= expression

3.7.1:
discriminant_constraint ::=
  (discriminant_association {, discriminant_association})

3.7.1:
discriminant_association ::=
  [*discriminant*_selector_name {'|' *discriminant*_selector_name} =>] expression

3.8:
record_type_definition ::= [[**abstract**] **tagged**] [**limited**] record_definition

3.8:
record_definition ::=
   record
     component_list
   end record [*record*_identifier]
 | **null record**

3.8:
component_list ::=
     component_item {component_item}
  | {component_item} variant_part
  | **null**;

3.8:
component_item ::= component_declaration | aspect_clause

3.8:
component_declaration ::=
  defining_identifier_list : component_definition [:= default_expression]
      [aspect_specification];

3.8.1:
variant_part ::=
  case *discriminant*_direct_name **is**
     variant
    {variant}
  end case;

3.8.1:
variant ::=
  when discrete_choice_list =>
    component_list

3.8.1:
discrete_choice_list ::= discrete_choice {'|' discrete_choice}

3.8.1:
discrete_choice ::=
  choice_expression | *discrete*_subtype_indication | range | **others**

3.9.1:
record_extension_part ::= **with** record_definition

3.9.3:
abstract_subprogram_declaration ::=
   [overriding_indicator]
   subprogram_specification **is abstract**
      [aspect_specification];

3.9.4:
interface_type_definition ::=
   [**limited** | **task** | **protected** | **synchronized**] **interface** [**and** interface_list]

3.9.4:
interface_list ::= *interface*_subtype_mark {**and** *interface*_subtype_mark}

3.10:
access_type_definition ::=
   [null_exclusion] access_to_object_definition
 | [null_exclusion] access_to_subprogram_definition

3.10:
access_to_object_definition ::=
   access [general_access_modifier] subtype_indication

3.10:
general_access_modifier ::= **all** | **constant**

3.10:
access_to_subprogram_definition ::=
   access [**protected**] **procedure** parameter_profile
 | **access** [**protected**] **function** parameter_and_result_profile

3.10:
null_exclusion ::= **not null**

3.10:
access_definition ::=
   [null_exclusion] **access** [**constant**] subtype_mark
 | [null_exclusion] **access** [**protected**] **procedure** parameter_profile
 | [null_exclusion] **access** [**protected**] **function** parameter_and_result_profile

3.10.1:
incomplete_type_declaration ::=
  type defining_identifier [discriminant_part] [**is tagged**];

3.11:
declarative_part ::= {declarative_item}

3.11:
declarative_item ::=
   basic_declarative_item | body

3.11:
basic_declarative_item ::=
   basic_declaration | aspect_clause | use_clause

3.11:
body ::= proper_body | body_stub

3.11:
proper_body ::=
   subprogram_body | package_body | task_body | protected_body

4.1:
name ::=
    direct_name | explicit_dereference
  | indexed_component | slice
  | selected_component | attribute_reference
  | type_conversion | function_call
  | character_literal | qualified_expression
  | generalized_reference | generalized_indexing
  | target_name

4.1:
direct_name ::= identifier | operator_symbol

4.1:
prefix ::= name | implicit_dereference

4.1:
explicit_dereference ::= name.**all**

4.1:
implicit_dereference ::= name

4.1.1:
indexed_component ::= prefix(expression {, expression})

4.1.2:
slice ::= prefix(discrete_range)

4.1.3:
selected_component ::= prefix . selector_name

4.1.3:
selector_name ::= identifier | character_literal | operator_symbol

4.1.4:
attribute_reference ::=
    prefix'attribute_designator
  | reduction_attribute_reference

4.1.4:
attribute_designator ::=
    identifier[(*static*_expression)]
  | Access | Delta | Digits | Mod

4.1.4:
range_attribute_reference ::= prefix'range_attribute_designator

4.1.4:
range_attribute_designator ::= Range[(*static*_expression)]

4.1.5:
generalized_reference ::= *reference_object*_name

4.1.6:
generalized_indexing ::= *indexable_container_object*_prefix actual_parameter_part

4.3:
aggregate ::=
    record_aggregate | extension_aggregate | array_aggregate
  | delta_aggregate | container_aggregate

4.3.1:
record_aggregate ::= (record_component_association_list)

4.3.1:
record_component_association_list ::=
    record_component_association {, record_component_association}
  | **null record**

4.3.1:
record_component_association ::=
   [component_choice_list =>] expression
  | component_choice_list => <>

4.3.1:
component_choice_list ::=
    *component*_selector_name {'|' *component*_selector_name}
  | **others**

4.3.2:
extension_aggregate ::=
   (ancestor_part **with** record_component_association_list)

4.3.2:
ancestor_part ::= expression | subtype_mark

4.3.3:
array_aggregate ::=
   positional_array_aggregate | null_array_aggregate | named_array_aggregate

4.3.3:
positional_array_aggregate ::=
   (expression, expression {, expression})
  | (expression {, expression}, **others** => expression)
  | (expression {, expression}, **others** => <>)
  | '[' expression {, expression}[, **others** => expression] ']'
  | '[' expression {, expression}, **others** => <> ']'

4.3.3:
null_array_aggregate ::= '[' ']'

4.3.3:
named_array_aggregate ::=
   (array_component_association_list)
  | '[' array_component_association_list ']'

4.3.3:
array_component_association_list ::=
   array_component_association {, array_component_association}

4.3.3:
array_component_association ::=
   discrete_choice_list => expression
  | discrete_choice_list => <>
  | iterated_component_association

4.3.3:
iterated_component_association ::=
   for defining_identifier **in** discrete_choice_list => expression
  | **for** iterator_specification => expression

4.3.4:
delta_aggregate ::= record_delta_aggregate | array_delta_aggregate

4.3.4:
record_delta_aggregate ::=
   (*base*_expression **with delta** record_component_association_list)

4.3.4:
array_delta_aggregate ::=
   (*base*_expression **with delta** array_component_association_list)
  | '[' *base*_expression **with delta** array_component_association_list ']'

4.3.5:
container_aggregate ::=
   null_container_aggregate
 | positional_container_aggregate
 | named_container_aggregate

4.3.5:
null_container_aggregate ::= '[' ']'

4.3.5:
positional_container_aggregate ::= '[' expression{, expression} ']'

4.3.5:
named_container_aggregate ::= '[' container_element_association_list ']'

4.3.5:
container_element_association_list ::=
   container_element_association {, container_element_association}

4.3.5:
container_element_association ::=
   key_choice_list => expression
 | key_choice_list => <>
 | iterated_element_association

4.3.5:
key_choice_list ::= key_choice {'|' key_choice}

4.3.5:
key_choice ::= *key*_expression | discrete_range

4.3.5:
iterated_element_association ::=
   for loop_parameter_specification[**use** *key*_expression] => expression
 | **for** iterator_specification[**use** *key*_expression] => expression

4.4:
expression ::=
    relation {**and** relation} | relation {**and then** relation}
  | relation {**or** relation} | relation {**or else** relation}
  | relation {**xor** relation}

4.4:
choice_expression ::=
    choice_relation {**and** choice_relation}
  | choice_relation {**or** choice_relation}
  | choice_relation {**xor** choice_relation}
  | choice_relation {**and then** choice_relation}
  | choice_relation {**or else** choice_relation}

4.4:
choice_relation ::=
    simple_expression [relational_operator simple_expression]

4.4:
relation ::=
    simple_expression [relational_operator simple_expression]
  | *tested*_simple_expression [**not**] **in** membership_choice_list
  | raise_expression

4.4:
membership_choice_list ::= membership_choice {'|' membership_choice}

4.4:
membership_choice ::= *choice*_simple_expression | range | subtype_mark

4.4:
simple_expression ::=
   [unary_adding_operator] term {binary_adding_operator term}

4.4:
term ::= factor {multiplying_operator factor}

4.4:
factor ::= primary [** primary] | **abs** primary | **not** primary

4.4:
primary ::=
   numeric_literal | **null** | string_literal | aggregate
 | name | allocator | (expression)
 | (conditional_expression) | (quantified_expression)
 | (declare_expression)

4.5:
logical_operator ::= **and** | **or** | **xor**

4.5:
relational_operator ::= = | /= | < | <= | > | >=

4.5:
binary_adding_operator ::= + | – | &

4.5:
unary_adding_operator ::= + | –

4.5:
multiplying_operator ::= * | / | **mod** | **rem**

4.5:
highest_precedence_operator ::= ** | **abs** | **not**

4.5.7:
conditional_expression ::= if_expression | case_expression

4.5.7:
if_expression ::=
  if condition **then** *dependent*_expression
  {**elsif** condition **then** *dependent*_expression}
  [**else** *dependent*_expression]

4.5.7:
condition ::= *boolean*_expression

4.5.7:
case_expression ::=
   case *selecting*_expression **is**
   case_expression_alternative {,
   case_expression_alternative}

4.5.7:
case_expression_alternative ::=
   when discrete_choice_list =>
      *dependent*_expression

4.5.8:
quantified_expression ::=
   for quantifier loop_parameter_specification => predicate
 | **for** quantifier iterator_specification => predicate

4.5.8:
quantifier ::= **all** | **some**

4.5.8:
predicate ::= *boolean*_expression

4.5.9:
declare_expression ::=
  declare {declare_item}
  begin *body*_expression

4.5.9:
declare_item ::= object_declaration | object_renaming_declaration

4.5.10:
reduction_attribute_reference ::=
  value_sequence'reduction_attribute_designator
 | prefix'reduction_attribute_designator

4.5.10:
value_sequence ::=
  '[' [**parallel**[(chunk_specification)] [aspect_specification]]
    iterated_element_association ']'

4.5.10:
reduction_attribute_designator ::=
  *reduction*_identifier(reduction_specification)

4.5.10:
reduction_specification ::= *reducer*_name, *initial_value*_expression
 | [Reducer =>]*reducer*_name, Initial_Value => *initial_value*_expression

4.6:
type_conversion ::=
  subtype_mark(expression)
 | subtype_mark(name)

4.7:
qualified_expression ::=
  subtype_mark'(expression) | subtype_mark'aggregate

4.8:
allocator ::=
  new [subpool_specification] subtype_indication
 | **new** [subpool_specification] qualified_expression

4.8:
subpool_specification ::= (*subpool_handle*_name)

5.1:
sequence_of_statements ::= statement {statement} {label}

5.1:
statement ::=
  {label} simple_statement | {label} compound_statement

5.1:
simple_statement ::= null_statement
  | assignment_statement | exit_statement
  | goto_statement | procedure_call_statement
  | simple_return_statement | entry_call_statement
  | requeue_statement | delay_statement
  | abort_statement | raise_statement
  | code_statement

5.1:
compound_statement ::=
    if_statement | case_statement
  | loop_statement | block_statement
  | extended_return_statement
  | parallel_block_statement
  | accept_statement | select_statement

5.1:
null_statement ::= **null**;

5.1:
label ::= <<*label*_statement_identifier>>

5.1:
statement_identifier ::= direct_name

5.2:
assignment_statement ::=
  *variable*_name := expression;

5.2.1:
target_name ::= @

5.3:
if_statement ::=
   if condition **then**
     sequence_of_statements
  {**elsif** condition **then**
     sequence_of_statements}
  [**else**
     sequence_of_statements]
   end if;

5.4:
case_statement ::=
  case *selecting*_expression **is**
     case_statement_alternative
    {case_statement_alternative}
  end case;

5.4:
case_statement_alternative ::=
  when discrete_choice_list =>
    sequence_of_statements

5.5:
loop_statement ::=
  [*loop*_statement_identifier:]
    [iteration_scheme] **loop**
      sequence_of_statements
    end loop [*loop*_identifier];

5.5:
iteration_scheme ::= **while** condition
  | **for** loop_parameter_specification
  | **for** iterator_specification
  | [**parallel** [aspect_specification]]
    for procedural_iterator
  | **parallel** [(chunk_specification)] [aspect_specification]
    for loop_parameter_specification
  | **parallel** [(chunk_specification)] [aspect_specification]
    for iterator_specification

5.5:
chunk_specification ::=
    *integer*_simple_expression
  | defining_identifier **in** discrete_subtype_definition

5.5:
loop_parameter_specification ::=
  defining_identifier **in** [**reverse**] discrete_subtype_definition
    [iterator_filter]

5.5:
iterator_filter ::= **when** condition

5.5.2:
iterator_specification ::=
    defining_identifier [: loop_parameter_subtype_indication] **in** [**reverse**] *iterator*_name
      [iterator_filter]
  | defining_identifier [: loop_parameter_subtype_indication] **of** [**reverse**] *iterable*_name
      [iterator_filter]

5.5.2:
loop_parameter_subtype_indication ::= subtype_indication | access_definition

5.5.3:
procedural_iterator ::=
    iterator_parameter_specification **of** iterator_procedure_call
      [iterator_filter]

5.5.3:
iterator_parameter_specification ::=
    formal_part
  | (defining_identifier{, defining_identifier})

5.5.3:
iterator_procedure_call ::=
    *procedure*_name
  | *procedure*_prefix iterator_actual_parameter_part

5.5.3:
iterator_actual_parameter_part ::=
    (iterator_parameter_association {, iterator_parameter_association})

5.5.3:
iterator_parameter_association ::=
    parameter_association
  | parameter_association_with_box

5.5.3:
parameter_association_with_box ::=
  [*formal_parameter*_selector_name =>] <>

5.6:
block_statement ::=
  [*block*_statement_identifier:]
      [**declare**
          declarative_part]
       begin
          handled_sequence_of_statements
       end [*block*_identifier];

5.6.1:
parallel_block_statement ::=
    parallel [(chunk_specification)] [aspect_specification] **do**
      sequence_of_statements
    and
      sequence_of_statements
    {**and**
      sequence_of_statements}
    end do;

5.7:
exit_statement ::=
  exit [*loop*_name] [**when** condition];

5.8:
goto_statement ::= **goto** *label*_name;

6.1:
subprogram_declaration ::=
   [overriding_indicator]
   subprogram_specification
      [aspect_specification];

6.1:
subprogram_specification ::=
   procedure_specification
 | function_specification

6.1:
procedure_specification ::=
   procedure defining_program_unit_name parameter_profile

6.1:
function_specification ::=
   function defining_designator parameter_and_result_profile

6.1:
designator ::= [parent_unit_name .]identifier | operator_symbol

6.1:
defining_designator ::=
   defining_program_unit_name | defining_operator_symbol

6.1:
defining_program_unit_name ::= [parent_unit_name .]defining_identifier

6.1:
operator_symbol ::= string_literal

6.1:
defining_operator_symbol ::= operator_symbol

6.1:
parameter_profile ::= [formal_part]

6.1:
parameter_and_result_profile ::=
   [formal_part] **return** [null_exclusion] subtype_mark
 | [formal_part] **return** access_definition

6.1:
formal_part ::=
  (parameter_specification {; parameter_specification})

6.1:
parameter_specification ::=
   defining_identifier_list : [**aliased**] mode [null_exclusion] subtype_mark [:= default_expression]
      [aspect_specification]
 | defining_identifier_list : access_definition [:= default_expression]
      [aspect_specification]

6.1:
mode ::= [**in**] | **in out** | **out**

6.1.2:
global_aspect_definition ::=
   null
 | Unspecified
 | global_mode global_designator
 | (global_aspect_element{; global_aspect_element})

6.1.2:
global_aspect_element ::=
   global_mode global_set
 | global_mode **all**
 | global_mode **synchronized**

6.1.2:
global_mode ::=
   basic_global_mode
 | extended_global_mode

6.1.2:
basic_global_mode ::= **in** | **in out** | **out**

6.1.2:
global_set ::= global_name {, global_name}

6.1.2:
global_designator ::= **all** | **synchronized** | global_name

6.1.2:
global_name ::= *object*_name | *package*_name

6.3:
subprogram_body ::=
   [overriding_indicator]
   subprogram_specification
      [aspect_specification] **is**
      declarative_part
   begin
      handled_sequence_of_statements
   end [designator];

6.4:
procedure_call_statement ::=
   *procedure*_name;
 | *procedure*_prefix actual_parameter_part;

6.4:
function_call ::=
   *function*_name
 | *function*_prefix actual_parameter_part

6.4:
actual_parameter_part ::=
   (parameter_association {, parameter_association})

6.4:
parameter_association ::=
  [*formal_parameter*_selector_name =>] explicit_actual_parameter

6.4:
explicit_actual_parameter ::= expression | *variable*_name

6.5:
simple_return_statement ::= **return** [expression];

6.5:
extended_return_object_declaration ::=
   defining_identifier : [**aliased**] [**constant**] return_subtype_indication [:= expression]
      [aspect_specification]

6.5:
extended_return_statement ::=
   return extended_return_object_declaration [**do**
      handled_sequence_of_statements
   end return];

6.5:
return_subtype_indication ::= subtype_indication | access_definition

6.7:
null_procedure_declaration ::=
  [overriding_indicator]
  procedure_specification **is null**
    [aspect_specification];

6.8:
expression_function_declaration ::=
  [overriding_indicator]
  function_specification **is**
    (expression)
    [aspect_specification];
| [overriding_indicator]
  function_specification **is**
    aggregate
    [aspect_specification];

7.1:
package_declaration ::= package_specification;

7.1:
package_specification ::=
   package defining_program_unit_name
     [aspect_specification] **is**
    {basic_declarative_item}
  [**private**
    {basic_declarative_item}]
  end [[parent_unit_name.]identifier]

7.2:
package_body ::=
   package body defining_program_unit_name
     [aspect_specification] **is**
    declarative_part
  [**begin**
     handled_sequence_of_statements]
  end [[parent_unit_name.]identifier];

7.3:
private_type_declaration ::=
  type defining_identifier [discriminant_part] **is** [[**abstract**] **tagged**] [**limited**] **private**
    [aspect_specification];

7.3:
private_extension_declaration ::=
  type defining_identifier [discriminant_part] **is**
    [**abstract**] [**limited** | **synchronized**] **new** *ancestor*_subtype_indication
    [**and** interface_list] **with private**
      [aspect_specification];

8.3.1:
overriding_indicator ::= [**not**] **overriding**

8.4:
use_clause ::= use_package_clause | use_type_clause

8.4:
use_package_clause ::= **use** *package*_name {, *package*_name};

8.4:
use_type_clause ::= **use** [**all**] **type** subtype_mark {, subtype_mark};

8.5:
renaming_declaration ::=
    object_renaming_declaration
  | exception_renaming_declaration
  | package_renaming_declaration
  | subprogram_renaming_declaration
  | generic_renaming_declaration

8.5.1:
object_renaming_declaration ::=
   defining_identifier [: [null_exclusion] subtype_mark] **renames** *object*_name
      [aspect_specification];
 | defining_identifier : access_definition **renames** *object*_name
      [aspect_specification];

8.5.2:
exception_renaming_declaration ::=
   defining_identifier : **exception renames** *exception*_name
     [aspect_specification];

8.5.3:
package_renaming_declaration ::=
   package defining_program_unit_name **renames** *package*_name
     [aspect_specification];

8.5.4:
subprogram_renaming_declaration ::=
   [overriding_indicator]
   subprogram_specification **renames** *callable_entity*_name
      [aspect_specification];

8.5.5:
generic_renaming_declaration ::=
   generic package defining_program_unit_name **renames** *generic_package*_name
      [aspect_specification];
 | **generic procedure** defining_program_unit_name **renames** *generic_procedure*_name
      [aspect_specification];
 | **generic function** defining_program_unit_name **renames** *generic_function*_name
      [aspect_specification];

9.1:
task_type_declaration ::=
  task type defining_identifier [known_discriminant_part]
      [aspect_specification] [**is**
    [**new** interface_list **with**]
    task_definition];

9.1:
single_task_declaration ::=
  task defining_identifier
      [aspect_specification] [**is**
    [**new** interface_list **with**]
    task_definition];

9.1:
task_definition ::=
    {task_item}
 [**private**
    {task_item}]
 end [*task*_identifier]

9.1:
task_item ::= entry_declaration | aspect_clause

9.1:
task_body ::=
  task body defining_identifier
      [aspect_specification] **is**
    declarative_part
  begin
    handled_sequence_of_statements
  end [*task*_identifier];

9.4:
protected_type_declaration ::=
  protected type defining_identifier [known_discriminant_part]
      [aspect_specification] **is**
    [**new** interface_list **with**]
    protected_definition;

9.4:
single_protected_declaration ::=
  protected defining_identifier
      [aspect_specification] **is**
    [**new** interface_list **with**]
    protected_definition;

9.4:
protected_definition ::=
    { protected_operation_declaration }
[**private**
    { protected_element_declaration }]
  end [*protected*_identifier]

9.4:
protected_operation_declaration ::= subprogram_declaration
    | entry_declaration
    | aspect_clause

9.4:
protected_element_declaration ::= protected_operation_declaration
    | component_declaration

9.4:
protected_body ::=
  protected body defining_identifier
      [aspect_specification] **is**
   { protected_operation_item }
  end [*protected*_identifier];

9.4:
protected_operation_item ::= subprogram_declaration
    | subprogram_body
    | null_procedure_declaration
    | expression_function_declaration
    | entry_body
    | aspect_clause

9.5:
synchronization_kind ::= By_Entry | By_Protected_Procedure | Optional

9.5.2:
entry_declaration ::=
  [overriding_indicator]
  entry defining_identifier [(discrete_subtype_definition)] parameter_profile
     [aspect_specification];

9.5.2:
accept_statement ::=
  accept *entry*_direct_name [(entry_index)] parameter_profile [**do**
    handled_sequence_of_statements
  end [*entry*_identifier]];

9.5.2:
entry_index ::= expression

9.5.2:
entry_body ::=
   entry defining_identifier entry_body_formal_part
     [aspect_specification]
   entry_barrier **is**
     declarative_part
   begin
     handled_sequence_of_statements
   end [*entry*_identifier];

9.5.2:
entry_body_formal_part ::= [(entry_index_specification)] parameter_profile

9.5.2:
entry_barrier ::= **when** condition

9.5.2:
entry_index_specification ::=
   for defining_identifier **in** discrete_subtype_definition [aspect_specification]

9.5.3:
entry_call_statement ::= *entry*_name [actual_parameter_part];

9.5.4:
requeue_statement ::= **requeue** *procedure_or_entry*_name [**with abort**];

9.6:
delay_statement ::= delay_until_statement | delay_relative_statement

9.6:
delay_until_statement ::= **delay until** *delay*_expression;

9.6:
delay_relative_statement ::= **delay** *delay*_expression;

9.7:
select_statement ::=
  selective_accept
 | timed_entry_call
 | conditional_entry_call
 | asynchronous_select

9.7.1:
selective_accept ::=
 select
  [guard]
   select_alternative
{ **or**
  [guard]
   select_alternative }
[**else**
  sequence_of_statements]
 end select;

9.7.1:
guard ::= **when** condition =>

9.7.1:
select_alternative ::=
  accept_alternative
 | delay_alternative
 | terminate_alternative

9.7.1:
accept_alternative ::=
 accept_statement [sequence_of_statements]

9.7.1:
delay_alternative ::=
 delay_statement [sequence_of_statements]

9.7.1:
terminate_alternative ::= **terminate**;

9.7.2:
timed_entry_call ::=
 select
  entry_call_alternative
 or
  delay_alternative
 end select;

9.7.2:
entry_call_alternative ::=
 procedure_or_entry_call [sequence_of_statements]

9.7.2:
procedure_or_entry_call ::=
 procedure_call_statement | entry_call_statement

9.7.3:
conditional_entry_call ::=
 select
  entry_call_alternative
 else
  sequence_of_statements
 end select;

9.7.4:
asynchronous_select ::=
 select
  triggering_alternative
 then abort
  abortable_part
 end select;

9.7.4:
triggering_alternative ::= triggering_statement [sequence_of_statements]

9.7.4:
triggering_statement ::= procedure_or_entry_call | delay_statement

9.7.4:
abortable_part ::= sequence_of_statements

9.8:
abort_statement ::= **abort** *task*_name {, *task*_name};

10.1.1:
compilation ::= {compilation_unit}

10.1.1:
compilation_unit ::=
   context_clause library_item
 | context_clause subunit

10.1.1:
library_item ::= [**private**] library_unit_declaration
 | library_unit_body
 | [**private**] library_unit_renaming_declaration

10.1.1:
library_unit_declaration ::=
    subprogram_declaration | package_declaration
  | generic_declaration | generic_instantiation

10.1.1:
library_unit_renaming_declaration ::=
  package_renaming_declaration
| generic_renaming_declaration
| subprogram_renaming_declaration

10.1.1:
library_unit_body ::= subprogram_body | package_body

10.1.1:
parent_unit_name ::= name

10.1.2:
context_clause ::= {context_item}

10.1.2:
context_item ::= with_clause | use_clause

10.1.2:
with_clause ::= limited_with_clause | nonlimited_with_clause

10.1.2:
limited_with_clause ::= **limited** [**private**] **with** *library_unit*_name {, *library_unit*_name};

10.1.2:
nonlimited_with_clause ::= [**private**] **with** *library_unit*_name {, *library_unit*_name};

10.1.3:
body_stub ::=
   subprogram_body_stub | package_body_stub
 | task_body_stub | protected_body_stub

10.1.3:
subprogram_body_stub ::=
   [overriding_indicator]
   subprogram_specification **is separate**
     [aspect_specification];

10.1.3:
package_body_stub ::=
  package body defining_identifier **is separate**
    [aspect_specification];

10.1.3:
task_body_stub ::=
  task body defining_identifier **is separate**
    [aspect_specification];

10.1.3:
protected_body_stub ::=
  protected body defining_identifier **is separate**
    [aspect_specification];

10.1.3:
subunit ::= **separate** (parent_unit_name) proper_body

11.1:
exception_declaration ::= defining_identifier_list : **exception**
  [aspect_specification];

11.2:
handled_sequence_of_statements ::=
    sequence_of_statements
  [**exception**
    exception_handler
   {exception_handler}]

11.2:
exception_handler ::=
  when [choice_parameter_specification:] exception_choice {'|' exception_choice} =>
    sequence_of_statements

11.2:
choice_parameter_specification ::= defining_identifier

11.2:
exception_choice ::= *exception*_name | **others**

11.3:
raise_statement ::= **raise**;
     | **raise** *exception*_name [**with** *string*_expression];

11.3:
raise_expression ::= **raise** *exception*_name [**with** *string*_simple_expression]

12.1:
generic_declaration ::=
   generic_subprogram_declaration | generic_package_declaration

12.1:
generic_subprogram_declaration ::=
   generic_formal_part subprogram_specification
     [aspect_specification];

12.1:
generic_package_declaration ::=
   generic_formal_part package_specification;

12.1:
generic_formal_part ::=
   generic {generic_formal_parameter_declaration | use_clause}

12.1:
generic_formal_parameter_declaration ::=
   formal_object_declaration
  | formal_type_declaration
  | formal_subprogram_declaration
  | formal_package_declaration

12.3:
generic_instantiation ::=
    package defining_program_unit_name **is**
       new *generic_package_*name [generic_actual_part]
         [aspect_specification];
  | [overriding_indicator]
    procedure defining_program_unit_name **is**
       new *generic_procedure_*name [generic_actual_part]
         [aspect_specification];
  | [overriding_indicator]
    function defining_designator **is**
       new *generic_function_*name [generic_actual_part]
         [aspect_specification];

12.3:
generic_actual_part ::=
  (generic_association {, generic_association})

12.3:
generic_association ::=
  [*generic_formal_parameter_*selector_name =>] explicit_generic_actual_parameter

12.3:
explicit_generic_actual_parameter ::= expression | *variable_*name
  | *subprogram_*name | *entry_*name | subtype_mark
  | *package_instance_*name

12.4:
formal_object_declaration ::=
    defining_identifier_list : mode [null_exclusion] subtype_mark [:= default_expression]
       [aspect_specification];
 | defining_identifier_list : mode access_definition [:= default_expression]
       [aspect_specification];

12.5:
formal_type_declaration ::=
     formal_complete_type_declaration
   | formal_incomplete_type_declaration

12.5:
formal_complete_type_declaration ::=
    type defining_identifier[discriminant_part] **is** formal_type_definition
       [**or use** *default_*subtype_mark] [aspect_specification];

12.5:
formal_incomplete_type_declaration ::=
    type defining_identifier[discriminant_part] [**is tagged**]
       [**or use** *default_*subtype_mark];

12.5:
formal_type_definition ::=
     formal_private_type_definition
   | formal_derived_type_definition
   | formal_discrete_type_definition
   | formal_signed_integer_type_definition
   | formal_modular_type_definition
   | formal_floating_point_definition
   | formal_ordinary_fixed_point_definition
   | formal_decimal_fixed_point_definition
   | formal_array_type_definition
   | formal_access_type_definition
   | formal_interface_type_definition

12.5.1:
formal_private_type_definition ::= [[**abstract**] **tagged**] [**limited**] **private**

12.5.1:
formal_derived_type_definition ::=
   [**abstract**] [**limited** | **synchronized**] **new** subtype_mark [[**and** interface_list] **with private**]

12.5.2:
formal_discrete_type_definition ::= (<>)

12.5.2:
formal_signed_integer_type_definition ::= **range** <>

12.5.2:
formal_modular_type_definition ::= **mod** <>

12.5.2:
formal_floating_point_definition ::= **digits** <>

12.5.2:
formal_ordinary_fixed_point_definition ::= **delta** <>

12.5.2:
formal_decimal_fixed_point_definition ::= **delta** <> **digits** <>

12.5.3:
formal_array_type_definition ::= array_type_definition

12.5.4:
formal_access_type_definition ::= access_type_definition

12.5.5:
formal_interface_type_definition ::= interface_type_definition

12.6:
formal_subprogram_declaration ::= formal_concrete_subprogram_declaration
  | formal_abstract_subprogram_declaration

12.6:
formal_concrete_subprogram_declaration ::=
   with subprogram_specification [**is** subprogram_default]
     [aspect_specification];

12.6:
formal_abstract_subprogram_declaration ::=
   with subprogram_specification **is abstract** [subprogram_default]
     [aspect_specification];

12.6:
subprogram_default ::= default_name | <> | **null**

12.6:
default_name ::= name

12.7:
formal_package_declaration ::=
   with package defining_identifier **is new** *generic_package_*name formal_package_actual_part
     [aspect_specification];

12.7:
formal_package_actual_part ::=
   ([**others** =>] <>)
 | [generic_actual_part]
 | (formal_package_association {, formal_package_association} [, **others** => <>])

12.7:
formal_package_association ::=
   generic_association
 | *generic_formal_parameter_*selector_name => <>

13.1:
aspect_clause ::= attribute_definition_clause
    | enumeration_representation_clause
    | record_representation_clause
    | at_clause

13.1:
local_name ::= direct_name
    | direct_name'attribute_designator
    | *library_unit*_name

13.1.1:
aspect_specification ::=
  with aspect_mark [=> aspect_definition] {,
       aspect_mark [=> aspect_definition] }

13.1.1:
aspect_mark ::= *aspect*_identifier['Class]

13.1.1:
aspect_definition ::=
   name | expression | identifier
 | aggregate | global_aspect_definition

13.3:
attribute_definition_clause ::=
     for local_name'attribute_designator **use** expression;
   | **for** local_name'attribute_designator **use** name;

13.4:
enumeration_representation_clause ::=
   for *first_subtype*_local_name **use** enumeration_aggregate;

13.4:
enumeration_aggregate ::= array_aggregate

13.5.1:
record_representation_clause ::=
   for *first_subtype*_local_name **use**
     record [mod_clause]
       {component_clause}
     end record [local_name];

13.5.1:
component_clause ::=
   *component*_local_name **at** position **range** first_bit .. last_bit;

13.5.1:
position ::= *static*_expression

13.5.1:
first_bit ::= *static*_simple_expression

13.5.1:
last_bit ::= *static*_simple_expression

13.8:
code_statement ::= qualified_expression;

13.11.3:
storage_pool_indicator ::= *storage_pool*_name | **null** | Standard

13.12:
restriction ::= *restriction*_identifier
   | *restriction_parameter*_identifier => restriction_parameter_argument

13.12:
restriction_parameter_argument ::= name | expression

H.7:
extended_global_mode ::=
  overriding basic_global_mode

H.7.1:
formal_parameter_set ::=
  formal_group_designator
 | formal_parameter_name
 | (formal_parameter_name{, formal_parameter_name})

H.7.1:
formal_group_designator ::= **null** | **all**

H.7.1:
formal_parameter_name ::=
  *formal_*subtype_mark
 | *formal_subprogram_*name
 | *formal_access_to_subprogram_object_*name

H.7.1:
dispatching_operation_set ::=
  dispatching_operation_specifier
 | (dispatching_operation_specifier{, dispatching_operation_specifier})

H.7.1:
dispatching_operation_specifier ::=
  *dispatching_operation_*name (*object_*name)

J.3:
delta_constraint ::= **delta** *static_*simple_expression [range_constraint]

J.7:
at_clause ::= **for** direct_name **use at** expression;

J.8:
mod_clause ::= **at mod** *static_*expression;

P.2 Syntax Cross Reference

1/3 In the following syntax cross reference, each syntactic category is followed by the subclause number where it is defined. In addition, each syntactic category *S* is followed by a list of the categories that use *S* in their definitions. For example, the first listing below shows that abort_statement appears in the definition of simple_statement.

abort_statement 9.8
  simple_statement 5.1

abortable_part 9.7.4
  asynchronous_select 9.7.4

abstract_subprogram_declaration 3.9.3
  basic_declaration 3.1

accept_alternative 9.7.1
  select_alternative 9.7.1

accept_statement 9.5.2
  accept_alternative 9.7.1
  compound_statement 5.1

access_definition 3.10
  component_definition 3.6
  discriminant_specification 3.7
  formal_object_declaration 12.4
  loop_parameter_subtype_indication 5.5.2
  object_declaration 3.3.1
  object_renaming_declaration 8.5.1
  parameter_and_result_profile 6.1
  parameter_specification 6.1
  return_subtype_indication 6.5

access_to_object_definition 3.10
  access_type_definition 3.10

access_to_subprogram_definition 3.10
  access_type_definition 3.10

access_type_definition 3.10
  formal_access_type_definition 12.5.4
  type_definition 3.2.1

actual_parameter_part 6.4
  entry_call_statement 9.5.3
  function_call 6.4
  generalized_indexing 4.1.6
  procedure_call_statement 6.4

aggregate 4.3
  aspect_definition 13.1.1
  expression_function_declaration 6.8
  primary 4.4
  qualified_expression 4.7

allocator 4.8
  primary 4.4

ancestor_part 4.3.2
  extension_aggregate 4.3.2

array_aggregate 4.3.3
  aggregate 4.3
  enumeration_aggregate 13.4

array_component_association 4.3.3
  array_component_association_list 4.3.3

array_component_association_list 4.3.3
  array_delta_aggregate 4.3.4
  named_array_aggregate 4.3.3

array_delta_aggregate 4.3.4
  delta_aggregate 4.3.4

array_type_definition 3.6
  formal_array_type_definition 12.5.3
  object_declaration 3.3.1
  type_definition 3.2.1

aspect_clause 13.1
  basic_declarative_item 3.11
  component_item 3.8
  protected_operation_declaration 9.4
  protected_operation_item 9.4
  task_item 9.1

aspect_definition 13.1.1
  aspect_specification 13.1.1

aspect_mark 13.1.1
  aspect_specification 13.1.1
  pragma_argument_association 2.8

aspect_specification 13.1.1
  abstract_subprogram_declaration 3.9.3
  component_declaration 3.8
  discriminant_specification 3.7
  entry_body 9.5.2
  entry_declaration 9.5.2
  entry_index_specification 9.5.2
  exception_declaration 11.1
  exception_renaming_declaration 8.5.2
  expression_function_declaration 6.8
  extended_return_object_declaration 6.5
  formal_abstract_subprogram_declaration 12.6
  formal_complete_type_declaration 12.5
  formal_concrete_subprogram_declaration 12.6
  formal_object_declaration 12.4
  formal_package_declaration 12.7
  full_type_declaration 3.2.1
  generic_instantiation 12.3
  generic_renaming_declaration 8.5.5
  generic_subprogram_declaration 12.1
  iteration_scheme 5.5
  null_procedure_declaration 6.7
  object_declaration 3.3.1
  object_renaming_declaration 8.5.1
  package_body 7.2
  package_body_stub 10.1.3
  package_renaming_declaration 8.5.3
  package_specification 7.1
  parallel_block_statement 5.6.1
  parameter_specification 6.1
  private_extension_declaration 7.3
  private_type_declaration 7.3
  protected_body 9.4
  protected_body_stub 10.1.3
  protected_type_declaration 9.4
  single_protected_declaration 9.4
  single_task_declaration 9.1
  subprogram_body 6.3
  subprogram_body_stub 10.1.3
  subprogram_declaration 6.1
  subprogram_renaming_declaration 8.5.4
  subtype_declaration 3.2.2
  task_body 9.1
  task_body_stub 10.1.3
  task_type_declaration 9.1
  value_sequence 4.5.10

assignment_statement 5.2
  simple_statement 5.1

asynchronous_select 9.7.4
  select_statement 9.7

at_clause J.7
  aspect_clause 13.1

attribute_definition_clause 13.3
  aspect_clause 13.1

attribute_designator 4.1.4
  attribute_definition_clause 13.3
  attribute_reference 4.1.4
  local_name 13.1

attribute_reference 4.1.4
  name 4.1

base 2.4.2
  based_literal 2.4.2

based_literal 2.4.2
  numeric_literal 2.4

based_numeral 2.4.2
  based_literal 2.4.2

basic_declaration 3.1
  basic_declarative_item 3.11

basic_declarative_item 3.11
  declarative_item 3.11
  package_specification 7.1

basic_global_mode 6.1.2
  extended_global_mode H.7
  global_mode 6.1.2

binary_adding_operator 4.5
  simple_expression 4.4

block_statement 5.6
  compound_statement 5.1

body 3.11
  declarative_item 3.11

body_stub 10.1.3
  body 3.11

case_expression 4.5.7
  conditional_expression 4.5.7

case_expression_alternative 4.5.7
  case_expression 4.5.7

case_statement 5.4
  compound_statement 5.1

case_statement_alternative 5.4
  case_statement 5.4

character 2.1
  comment 2.7

character_literal 2.5
  defining_character_literal 3.5.1
  name 4.1
  selector_name 4.1.3

choice_expression 4.4
  discrete_choice 3.8.1

choice_parameter_specification 11.2
  exception_handler 11.2

choice_relation 4.4
  choice_expression 4.4

chunk_specification 5.5
  iteration_scheme 5.5
  parallel_block_statement 5.6.1
  value_sequence 4.5.10

code_statement 13.8
  simple_statement 5.1

compilation_unit 10.1.1
  compilation 10.1.1

component_choice_list 4.3.1
  record_component_association 4.3.1

component_clause 13.5.1
  record_representation_clause 13.5.1

component_declaration 3.8
  component_item 3.8
  protected_element_declaration 9.4

component_definition 3.6
  component_declaration 3.8
  constrained_array_definition 3.6
  unconstrained_array_definition 3.6

component_item 3.8
  component_list 3.8

component_list 3.8
  record_definition 3.8
  variant 3.8.1

composite_constraint 3.2.2
  constraint 3.2.2

compound_statement 5.1
  statement 5.1

condition 4.5.7
  entry_barrier 9.5.2
  exit_statement 5.7
  guard 9.7.1
  if_expression 4.5.7
  if_statement 5.3
  iteration_scheme 5.5
  iterator_filter 5.5

conditional_entry_call 9.7.3
  select_statement 9.7

conditional_expression 4.5.7
  primary 4.4

constrained_array_definition 3.6
  array_type_definition 3.6

constraint 3.2.2
    subtype_indication 3.2.2

container_aggregate 4.3.5
    aggregate 4.3

container_element_association 4.3.5
    container_element_association_list 4.3.5

container_element_association_list 4.3.5
    named_container_aggregate 4.3.5

context_clause 10.1.2
    compilation_unit 10.1.1

context_item 10.1.2
    context_clause 10.1.2

decimal_fixed_point_definition 3.5.9
    fixed_point_definition 3.5.9

decimal_literal 2.4.1
    numeric_literal 2.4

declarative_item 3.11
    declarative_part 3.11

declarative_part 3.11
    block_statement 5.6
    entry_body 9.5.2
    package_body 7.2
    subprogram_body 6.3
    task_body 9.1

declare_expression 4.5.9
    primary 4.4

declare_item 4.5.9
    declare_expression 4.5.9

default_expression 3.7
    component_declaration 3.8
    discriminant_specification 3.7
    formal_object_declaration 12.4
    parameter_specification 6.1

default_name 12.6
    subprogram_default 12.6

defining_character_literal 3.5.1
    enumeration_literal_specification 3.5.1

defining_designator 6.1
    function_specification 6.1
    generic_instantiation 12.3

defining_identifier 3.1
    choice_parameter_specification 11.2
    chunk_specification 5.5
    defining_identifier_list 3.3.1
    defining_program_unit_name 6.1
    entry_body 9.5.2
    entry_declaration 9.5.2
    entry_index_specification 9.5.2
    enumeration_literal_specification 3.5.1
    exception_renaming_declaration 8.5.2
    extended_return_object_declaration 6.5
    formal_complete_type_declaration 12.5
    formal_incomplete_type_declaration 12.5
    formal_package_declaration 12.7
    full_type_declaration 3.2.1
    incomplete_type_declaration 3.10.1
    iterated_component_association 4.3.3
    iterator_parameter_specification 5.5.3
    iterator_specification 5.5.2
    loop_parameter_specification 5.5
    object_renaming_declaration 8.5.1
    package_body_stub 10.1.3
    private_extension_declaration 7.3
    private_type_declaration 7.3
    protected_body 9.4
    protected_body_stub 10.1.3
    protected_type_declaration 9.4
    single_protected_declaration 9.4
    single_task_declaration 9.1
    subtype_declaration 3.2.2
    task_body 9.1
    task_body_stub 10.1.3
    task_type_declaration 9.1

defining_identifier_list 3.3.1
    component_declaration 3.8
    discriminant_specification 3.7
    exception_declaration 11.1
    formal_object_declaration 12.4
    number_declaration 3.3.2
    object_declaration 3.3.1
    parameter_specification 6.1

defining_operator_symbol 6.1
    defining_designator 6.1

defining_program_unit_name 6.1
    defining_designator 6.1
    generic_instantiation 12.3
    generic_renaming_declaration 8.5.5
    package_body 7.2
    package_renaming_declaration 8.5.3
    package_specification 7.1
    procedure_specification 6.1

delay_alternative 9.7.1
    select_alternative 9.7.1
    timed_entry_call 9.7.2

delay_relative_statement 9.6
    delay_statement 9.6

delay_statement 9.6
- delay_alternative 9.7.1
- simple_statement 5.1
- triggering_statement 9.7.4

delay_until_statement 9.6
- delay_statement 9.6

delta_aggregate 4.3.4
- aggregate 4.3

delta_constraint J.3
- scalar_constraint 3.2.2

derived_type_definition 3.4
- type_definition 3.2.1

designator 6.1
- subprogram_body 6.3

digit 2.4.1
- extended_digit 2.4.2
- numeral 2.4.1

digits_constraint 3.5.9
- scalar_constraint 3.2.2

direct_name 4.1
- accept_statement 9.5.2
- at_clause J.7
- local_name 13.1
- name 4.1
- statement_identifier 5.1
- variant_part 3.8.1

discrete_choice 3.8.1
- discrete_choice_list 3.8.1

discrete_choice_list 3.8.1
- array_component_association 4.3.3
- case_expression_alternative 4.5.7
- case_statement_alternative 5.4
- iterated_component_association 4.3.3
- variant 3.8.1

discrete_range 3.6.1
- index_constraint 3.6.1
- key_choice 4.3.5
- slice 4.1.2

discrete_subtype_definition 3.6
- chunk_specification 5.5
- constrained_array_definition 3.6
- entry_declaration 9.5.2
- entry_index_specification 9.5.2
- loop_parameter_specification 5.5

discriminant_association 3.7.1
- discriminant_constraint 3.7.1

discriminant_constraint 3.7.1
- composite_constraint 3.2.2

discriminant_part 3.7
- formal_complete_type_declaration 12.5
- formal_incomplete_type_declaration 12.5
- incomplete_type_declaration 3.10.1
- private_extension_declaration 7.3
- private_type_declaration 7.3

discriminant_specification 3.7
- known_discriminant_part 3.7

dispatching_operation_specifier H.7.1
- dispatching_operation_set H.7.1

entry_barrier 9.5.2
- entry_body 9.5.2

entry_body 9.5.2
- protected_operation_item 9.4

entry_body_formal_part 9.5.2
- entry_body 9.5.2

entry_call_alternative 9.7.2
- conditional_entry_call 9.7.3
- timed_entry_call 9.7.2

entry_call_statement 9.5.3
- procedure_or_entry_call 9.7.2
- simple_statement 5.1

entry_declaration 9.5.2
- protected_operation_declaration 9.4
- task_item 9.1

entry_index 9.5.2
- accept_statement 9.5.2

entry_index_specification 9.5.2
- entry_body_formal_part 9.5.2

enumeration_aggregate 13.4
- enumeration_representation_clause 13.4

enumeration_literal_specification 3.5.1
- enumeration_type_definition 3.5.1

enumeration_representation_clause 13.4
- aspect_clause 13.1

enumeration_type_definition 3.5.1
- type_definition 3.2.1

exception_choice 11.2
- exception_handler 11.2

exception_declaration 11.1
- basic_declaration 3.1

exception_handler 11.2
- handled_sequence_of_statements 11.2

exception_renaming_declaration 8.5.2
- renaming_declaration 8.5

exit_statement 5.7
simple_statement 5.1

explicit_actual_parameter 6.4
parameter_association 6.4

explicit_dereference 4.1
name 4.1

explicit_generic_actual_parameter 12.3
generic_association 12.3

exponent 2.4.1
based_literal 2.4.2
decimal_literal 2.4.1

expression 4.4
ancestor_part 4.3.2
array_component_association 4.3.3
array_delta_aggregate 4.3.4
aspect_definition 13.1.1
assignment_statement 5.2
at_clause J.7
attribute_definition_clause 13.3
attribute_designator 4.1.4
case_expression 4.5.7
case_expression_alternative 4.5.7
case_statement 5.4
condition 4.5.7
container_element_association 4.3.5
decimal_fixed_point_definition 3.5.9
declare_expression 4.5.9
default_expression 3.7
delay_relative_statement 9.6
delay_until_statement 9.6
discriminant_association 3.7.1
entry_index 9.5.2
explicit_actual_parameter 6.4
explicit_generic_actual_parameter 12.3
expression_function_declaration 6.8
extended_return_object_declaration 6.5
floating_point_definition 3.5.7
if_expression 4.5.7
indexed_component 4.1.1
iterated_component_association 4.3.3
iterated_element_association 4.3.5
key_choice 4.3.5
mod_clause J.8
modular_type_definition 3.5.4
number_declaration 3.3.2
object_declaration 3.3.1
ordinary_fixed_point_definition 3.5.9
position 13.5.1
positional_array_aggregate 4.3.3
positional_container_aggregate 4.3.5
pragma_argument_association 2.8
predicate 4.5.8
primary 4.4
qualified_expression 4.7
raise_statement 11.3
range_attribute_designator 4.1.4
record_component_association 4.3.1
record_delta_aggregate 4.3.4
reduction_specification 4.5.10
restriction_parameter_argument 13.12
simple_return_statement 6.5
type_conversion 4.6

expression_function_declaration 6.8
basic_declaration 3.1
protected_operation_item 9.4

extended_digit 2.4.2
based_numeral 2.4.2

extended_global_mode H.7
  global_mode 6.1.2

extended_return_object_declaration 6.5
  extended_return_statement 6.5

extended_return_statement 6.5
  compound_statement 5.1

extension_aggregate 4.3.2
  aggregate 4.3

factor 4.4
  term 4.4

first_bit 13.5.1
  component_clause 13.5.1

fixed_point_definition 3.5.9
  real_type_definition 3.5.6

floating_point_definition 3.5.7
  real_type_definition 3.5.6

formal_abstract_subprogram_declaration 12.6
  formal_subprogram_declaration 12.6

formal_access_type_definition 12.5.4
  formal_type_definition 12.5

formal_array_type_definition 12.5.3
  formal_type_definition 12.5

formal_complete_type_declaration 12.5
  formal_type_declaration 12.5

formal_concrete_subprogram_declaration 12.6
  formal_subprogram_declaration 12.6

formal_decimal_fixed_point_definition 12.5.2
  formal_type_definition 12.5

formal_derived_type_definition 12.5.1
  formal_type_definition 12.5

formal_discrete_type_definition 12.5.2
  formal_type_definition 12.5

formal_floating_point_definition 12.5.2
  formal_type_definition 12.5

formal_group_designator H.7.1
  formal_parameter_set H.7.1

formal_incomplete_type_declaration 12.5
  formal_type_declaration 12.5

formal_interface_type_definition 12.5.5
  formal_type_definition 12.5

formal_modular_type_definition 12.5.2
  formal_type_definition 12.5

formal_object_declaration 12.4
  generic_formal_parameter_declaration 12.1

formal_ordinary_fixed_point_definition 12.5.2
  formal_type_definition 12.5

formal_package_actual_part 12.7
  formal_package_declaration 12.7

formal_package_association 12.7
  formal_package_actual_part 12.7

formal_package_declaration 12.7
  generic_formal_parameter_declaration 12.1

formal_parameter_name H.7.1
  formal_parameter_set H.7.1

formal_part 6.1
  iterator_parameter_specification 5.5.3
  parameter_and_result_profile 6.1
  parameter_profile 6.1

formal_private_type_definition 12.5.1
  formal_type_definition 12.5

formal_signed_integer_type_definition 12.5.2
  formal_type_definition 12.5

formal_subprogram_declaration 12.6
  generic_formal_parameter_declaration 12.1

formal_type_declaration 12.5
  generic_formal_parameter_declaration 12.1

formal_type_definition 12.5
  formal_complete_type_declaration 12.5

full_type_declaration 3.2.1
  type_declaration 3.2.1

function_call 6.4
  name 4.1

function_specification 6.1
  expression_function_declaration 6.8
  subprogram_specification 6.1

general_access_modifier 3.10
  access_to_object_definition 3.10

generalized_indexing 4.1.6
  name 4.1

generalized_reference 4.1.5
  name 4.1

generic_actual_part 12.3
  formal_package_actual_part 12.7
  generic_instantiation 12.3

generic_association 12.3
  formal_package_association 12.7
  generic_actual_part 12.3

generic_declaration 12.1
basic_declaration 3.1
library_unit_declaration 10.1.1

generic_formal_parameter_declaration 12.1
generic_formal_part 12.1

generic_formal_part 12.1
generic_package_declaration 12.1
generic_subprogram_declaration 12.1

generic_instantiation 12.3
basic_declaration 3.1
library_unit_declaration 10.1.1

generic_package_declaration 12.1
generic_declaration 12.1

generic_renaming_declaration 8.5.5
library_unit_renaming_declaration 10.1.1
renaming_declaration 8.5

generic_subprogram_declaration 12.1
generic_declaration 12.1

global_aspect_definition 6.1.2
aspect_definition 13.1.1

global_aspect_element 6.1.2
global_aspect_definition 6.1.2

global_designator 6.1.2
global_aspect_definition 6.1.2

global_mode 6.1.2
global_aspect_definition 6.1.2
global_aspect_element 6.1.2

global_name 6.1.2
global_designator 6.1.2
global_set 6.1.2

global_set 6.1.2
global_aspect_element 6.1.2

goto_statement 5.8
simple_statement 5.1

graphic_character 2.1
character_literal 2.5
string_element 2.6

guard 9.7.1
selective_accept 9.7.1

handled_sequence_of_statements 11.2
accept_statement 9.5.2
block_statement 5.6
entry_body 9.5.2
extended_return_statement 6.5
package_body 7.2
subprogram_body 6.3
task_body 9.1

identifier 2.3
accept_statement 9.5.2
aspect_definition 13.1.1
aspect_mark 13.1.1
attribute_designator 4.1.4
block_statement 5.6
defining_identifier 3.1
designator 6.1
direct_name 4.1
entry_body 9.5.2
loop_statement 5.5
package_body 7.2
package_specification 7.1
pragma 2.8
pragma_argument_association 2.8
protected_body 9.4
protected_definition 9.4
record_definition 3.8
reduction_attribute_designator 4.5.10
restriction 13.12
selector_name 4.1.3
task_body 9.1
task_definition 9.1

identifier_extend 2.3
identifier 2.3

identifier_start 2.3
identifier 2.3

if_expression 4.5.7
conditional_expression 4.5.7

if_statement 5.3
compound_statement 5.1

implicit_dereference 4.1
prefix 4.1

incomplete_type_declaration 3.10.1
type_declaration 3.2.1

index_constraint 3.6.1
composite_constraint 3.2.2

index_subtype_definition 3.6
unconstrained_array_definition 3.6

indexed_component 4.1.1
name 4.1

integer_type_definition 3.5.4
type_definition 3.2.1

interface_list 3.9.4
derived_type_definition 3.4
formal_derived_type_definition 12.5.1
interface_type_definition 3.9.4
private_extension_declaration 7.3
protected_type_declaration 9.4
single_protected_declaration 9.4
single_task_declaration 9.1
task_type_declaration 9.1

interface_type_definition 3.9.4
formal_interface_type_definition 12.5.5
type_definition 3.2.1

iterated_component_association 4.3.3
array_component_association 4.3.3

iterated_element_association 4.3.5
container_element_association 4.3.5
value_sequence 4.5.10

iteration_scheme 5.5
loop_statement 5.5

iterator_actual_parameter_part 5.5.3
iterator_procedure_call 5.5.3

iterator_filter 5.5
iterator_specification 5.5.2
loop_parameter_specification 5.5
procedural_iterator 5.5.3

iterator_parameter_association 5.5.3
iterator_actual_parameter_part 5.5.3

iterator_parameter_specification 5.5.3
procedural_iterator 5.5.3

iterator_procedure_call 5.5.3
procedural_iterator 5.5.3

iterator_specification 5.5.2
iterated_component_association 4.3.3
iterated_element_association 4.3.5
iteration_scheme 5.5
quantified_expression 4.5.8

key_choice 4.3.5
key_choice_list 4.3.5

key_choice_list 4.3.5
container_element_association 4.3.5

known_discriminant_part 3.7
discriminant_part 3.7
full_type_declaration 3.2.1
protected_type_declaration 9.4
task_type_declaration 9.1

label 5.1
sequence_of_statements 5.1
statement 5.1

last_bit 13.5.1
component_clause 13.5.1

letter_lowercase ...
identifier_start 2.3

letter_modifier ...
identifier_start 2.3

letter_other ...
identifier_start 2.3

letter_titlecase ...
identifier_start 2.3

letter_uppercase ...
identifier_start 2.3

library_item 10.1.1
compilation_unit 10.1.1

library_unit_body 10.1.1
library_item 10.1.1

library_unit_declaration 10.1.1
library_item 10.1.1

library_unit_renaming_declaration 10.1.1
library_item 10.1.1

limited_with_clause 10.1.2
with_clause 10.1.2

local_name 13.1
attribute_definition_clause 13.3
component_clause 13.5.1
enumeration_representation_clause 13.4
record_representation_clause 13.5.1

loop_parameter_specification 5.5
iterated_element_association 4.3.5
iteration_scheme 5.5
quantified_expression 4.5.8

loop_parameter_subtype_indication 5.5.2
iterator_specification 5.5.2

loop_statement 5.5
compound_statement 5.1

mark_non_spacing ...
identifier_extend 2.3

mark_spacing_combining ...
identifier_extend 2.3

membership_choice 4.4
membership_choice_list 4.4

membership_choice_list 4.4
relation 4.4

mod_clause J.8
record_representation_clause 13.5.1

mode 6.1
formal_object_declaration 12.4
parameter_specification 6.1

modular_type_definition 3.5.4
integer_type_definition 3.5.4

multiplying_operator 4.5
term 4.4

name 4.1
abort_statement 9.8
aspect_definition 13.1.1
assignment_statement 5.2
attribute_definition_clause 13.3
default_name 12.6
dispatching_operation_specifier H.7.1
entry_call_statement 9.5.3
exception_choice 11.2
exception_renaming_declaration 8.5.2
exit_statement 5.7
explicit_actual_parameter 6.4
explicit_dereference 4.1
explicit_generic_actual_parameter 12.3
formal_package_declaration 12.7
formal_parameter_name H.7.1
function_call 6.4
generalized_reference 4.1.5
generic_instantiation 12.3
generic_renaming_declaration 8.5.5
global_name 6.1.2
goto_statement 5.8
implicit_dereference 4.1
iterator_procedure_call 5.5.3
iterator_specification 5.5.2
limited_with_clause 10.1.2
local_name 13.1
nonlimited_with_clause 10.1.2
object_renaming_declaration 8.5.1
package_renaming_declaration 8.5.3
parent_unit_name 10.1.1
pragma_argument_association 2.8
prefix 4.1
primary 4.4
procedure_call_statement 6.4
raise_expression 11.3
raise_statement 11.3
reduction_specification 4.5.10
requeue_statement 9.5.4
restriction_parameter_argument 13.12
storage_pool_indicator 13.11.3
subpool_specification 4.8
subprogram_renaming_declaration 8.5.4
subtype_mark 3.2.2
type_conversion 4.6
use_package_clause 8.4

named_array_aggregate 4.3.3
array_aggregate 4.3.3

named_container_aggregate 4.3.5
container_aggregate 4.3.5

nonlimited_with_clause 10.1.2
with_clause 10.1.2

null_array_aggregate 4.3.3
array_aggregate 4.3.3

null_container_aggregate 4.3.5
container_aggregate 4.3.5

null_exclusion 3.10
access_definition 3.10
access_type_definition 3.10
discriminant_specification 3.7
formal_object_declaration 12.4
object_renaming_declaration 8.5.1
parameter_and_result_profile 6.1
parameter_specification 6.1
subtype_indication 3.2.2

null_procedure_declaration 6.7
basic_declaration 3.1
protected_operation_item 9.4

null_statement 5.1
simple_statement 5.1

number_decimal ...
identifier_extend 2.3

number_declaration 3.3.2
basic_declaration 3.1

number_letter ...
identifier_start 2.3

numeral 2.4.1
base 2.4.2
decimal_literal 2.4.1
exponent 2.4.1

numeric_literal 2.4
primary 4.4

object_declaration 3.3.1
basic_declaration 3.1
declare_item 4.5.9

object_renaming_declaration 8.5.1
declare_item 4.5.9
renaming_declaration 8.5

operator_symbol 6.1
defining_operator_symbol 6.1
designator 6.1
direct_name 4.1
selector_name 4.1.3

ordinary_fixed_point_definition 3.5.9
fixed_point_definition 3.5.9

overriding_indicator 8.3.1
- abstract_subprogram_declaration 3.9.3
- entry_declaration 9.5.2
- expression_function_declaration 6.8
- generic_instantiation 12.3
- null_procedure_declaration 6.7
- subprogram_body 6.3
- subprogram_body_stub 10.1.3
- subprogram_declaration 6.1
- subprogram_renaming_declaration 8.5.4

package_body 7.2
- library_unit_body 10.1.1
- proper_body 3.11

package_body_stub 10.1.3
- body_stub 10.1.3

package_declaration 7.1
- basic_declaration 3.1
- library_unit_declaration 10.1.1

package_renaming_declaration 8.5.3
- library_unit_renaming_declaration 10.1.1
- renaming_declaration 8.5

package_specification 7.1
- generic_package_declaration 12.1
- package_declaration 7.1

parallel_block_statement 5.6.1
- compound_statement 5.1

parameter_and_result_profile 6.1
- access_definition 3.10
- access_to_subprogram_definition 3.10
- function_specification 6.1

parameter_association 6.4
- actual_parameter_part 6.4
- iterator_parameter_association 5.5.3

parameter_association_with_box 5.5.3
- iterator_parameter_association 5.5.3

parameter_profile 6.1
- accept_statement 9.5.2
- access_definition 3.10
- access_to_subprogram_definition 3.10
- entry_body_formal_part 9.5.2
- entry_declaration 9.5.2
- procedure_specification 6.1

parameter_specification 6.1
- formal_part 6.1

parent_unit_name 10.1.1
- defining_program_unit_name 6.1
- designator 6.1
- package_body 7.2
- package_specification 7.1
- subunit 10.1.3

position 13.5.1
- component_clause 13.5.1

positional_array_aggregate 4.3.3
- array_aggregate 4.3.3

positional_container_aggregate 4.3.5
- container_aggregate 4.3.5

pragma_argument_association 2.8
- pragma 2.8

predicate 4.5.8
- quantified_expression 4.5.8

prefix 4.1
- attribute_reference 4.1.4
- function_call 6.4
- generalized_indexing 4.1.6
- indexed_component 4.1.1
- iterator_procedure_call 5.5.3
- procedure_call_statement 6.4
- range_attribute_reference 4.1.4
- reduction_attribute_reference 4.5.10
- selected_component 4.1.3
- slice 4.1.2

primary 4.4
- factor 4.4

private_extension_declaration 7.3
- type_declaration 3.2.1

private_type_declaration 7.3
- type_declaration 3.2.1

procedural_iterator 5.5.3
- iteration_scheme 5.5

procedure_call_statement 6.4
- procedure_or_entry_call 9.7.2
- simple_statement 5.1

procedure_or_entry_call 9.7.2
- entry_call_alternative 9.7.2
- triggering_statement 9.7.4

procedure_specification 6.1
- null_procedure_declaration 6.7
- subprogram_specification 6.1

proper_body 3.11
- body 3.11
- subunit 10.1.3

protected_body 9.4
- proper_body 3.11

protected_body_stub 10.1.3
- body_stub 10.1.3

protected_definition 9.4
    protected_type_declaration 9.4
    single_protected_declaration 9.4

protected_element_declaration 9.4
    protected_definition 9.4

protected_operation_declaration 9.4
    protected_definition 9.4
    protected_element_declaration 9.4

protected_operation_item 9.4
    protected_body 9.4

protected_type_declaration 9.4
    full_type_declaration 3.2.1

punctuation_connector ...
    identifier_extend 2.3

qualified_expression 4.7
    allocator 4.8
    code_statement 13.8
    name 4.1

quantified_expression 4.5.8
    primary 4.4

quantifier 4.5.8
    quantified_expression 4.5.8

raise_expression 11.3
    relation 4.4

raise_statement 11.3
    simple_statement 5.1

range 3.5
    discrete_choice 3.8.1
    discrete_range 3.6.1
    discrete_subtype_definition 3.6
    membership_choice 4.4
    range_constraint 3.5

range_attribute_designator 4.1.4
    range_attribute_reference 4.1.4

range_attribute_reference 4.1.4
    range 3.5

range_constraint 3.5
    delta_constraint J.3
    digits_constraint 3.5.9
    scalar_constraint 3.2.2

real_range_specification 3.5.7
    decimal_fixed_point_definition 3.5.9
    floating_point_definition 3.5.7
    ordinary_fixed_point_definition 3.5.9

real_type_definition 3.5.6
    type_definition 3.2.1

record_aggregate 4.3.1
    aggregate 4.3

record_component_association 4.3.1
    record_component_association_list 4.3.1

record_component_association_list 4.3.1
    extension_aggregate 4.3.2
    record_aggregate 4.3.1
    record_delta_aggregate 4.3.4

record_definition 3.8
    record_extension_part 3.9.1
    record_type_definition 3.8

record_delta_aggregate 4.3.4
    delta_aggregate 4.3.4

record_extension_part 3.9.1
    derived_type_definition 3.4

record_representation_clause 13.5.1
    aspect_clause 13.1

record_type_definition 3.8
    type_definition 3.2.1

reduction_attribute_designator 4.5.10
    reduction_attribute_reference 4.5.10

reduction_attribute_reference 4.5.10
    attribute_reference 4.1.4

reduction_specification 4.5.10
    reduction_attribute_designator 4.5.10

relation 4.4
    expression 4.4

relational_operator 4.5
    choice_relation 4.4
    relation 4.4

renaming_declaration 8.5
    basic_declaration 3.1

requeue_statement 9.5.4
    simple_statement 5.1

restriction_parameter_argument 13.12
    restriction 13.12

return_subtype_indication 6.5
    extended_return_object_declaration 6.5

scalar_constraint 3.2.2
    constraint 3.2.2

select_alternative 9.7.1
    selective_accept 9.7.1

select_statement 9.7
    compound_statement 5.1

selected_component 4.1.3
    name 4.1

selective_accept 9.7.1
    select_statement 9.7

selector_name 4.1.3
    component_choice_list 4.3.1
    discriminant_association 3.7.1
    formal_package_association 12.7
    generic_association 12.3
    parameter_association 6.4
    parameter_association_with_box 5.5.3
    selected_component 4.1.3

sequence_of_statements 5.1
    abortable_part 9.7.4
    accept_alternative 9.7.1
    case_statement_alternative 5.4
    conditional_entry_call 9.7.3
    delay_alternative 9.7.1
    entry_call_alternative 9.7.2
    exception_handler 11.2
    handled_sequence_of_statements 11.2
    if_statement 5.3
    loop_statement 5.5
    parallel_block_statement 5.6.1
    selective_accept 9.7.1
    triggering_alternative 9.7.4

signed_integer_type_definition 3.5.4
    integer_type_definition 3.5.4

simple_expression 4.4
    choice_relation 4.4
    chunk_specification 5.5
    delta_constraint J.3
    digits_constraint 3.5.9
    first_bit 13.5.1
    last_bit 13.5.1
    membership_choice 4.4
    raise_expression 11.3
    range 3.5
    real_range_specification 3.5.7
    relation 4.4
    signed_integer_type_definition 3.5.4

simple_return_statement 6.5
    simple_statement 5.1

simple_statement 5.1
    statement 5.1

single_protected_declaration 9.4
    object_declaration 3.3.1

single_task_declaration 9.1
    object_declaration 3.3.1

slice 4.1.2
    name 4.1

statement 5.1
    sequence_of_statements 5.1

statement_identifier 5.1
    block_statement 5.6
    label 5.1
    loop_statement 5.5

string_element 2.6
    string_literal 2.6

string_literal 2.6
    operator_symbol 6.1
    primary 4.4

subpool_specification 4.8
    allocator 4.8

subprogram_body 6.3
    library_unit_body 10.1.1
    proper_body 3.11
    protected_operation_item 9.4

subprogram_body_stub 10.1.3
    body_stub 10.1.3

subprogram_declaration 6.1
    basic_declaration 3.1
    library_unit_declaration 10.1.1
    protected_operation_declaration 9.4
    protected_operation_item 9.4

subprogram_default 12.6
    formal_abstract_subprogram_declaration 12.6
    formal_concrete_subprogram_declaration 12.6

subprogram_renaming_declaration 8.5.4
    library_unit_renaming_declaration 10.1.1
    renaming_declaration 8.5

subprogram_specification 6.1
    abstract_subprogram_declaration 3.9.3
    formal_abstract_subprogram_declaration 12.6
    formal_concrete_subprogram_declaration 12.6
    generic_subprogram_declaration 12.1
    subprogram_body 6.3
    subprogram_body_stub 10.1.3
    subprogram_declaration 6.1
    subprogram_renaming_declaration 8.5.4

subtype_declaration 3.2.2
    basic_declaration 3.1

subtype_indication 3.2.2
- access_to_object_definition 3.10
- allocator 4.8
- component_definition 3.6
- derived_type_definition 3.4
- discrete_choice 3.8.1
- discrete_range 3.6.1
- discrete_subtype_definition 3.6
- loop_parameter_subtype_indication 5.5.2
- object_declaration 3.3.1
- private_extension_declaration 7.3
- return_subtype_indication 6.5
- subtype_declaration 3.2.2

subtype_mark 3.2.2
- access_definition 3.10
- ancestor_part 4.3.2
- discriminant_specification 3.7
- explicit_generic_actual_parameter 12.3
- formal_complete_type_declaration 12.5
- formal_derived_type_definition 12.5.1
- formal_incomplete_type_declaration 12.5
- formal_object_declaration 12.4
- formal_parameter_name H.7.1
- index_subtype_definition 3.6
- interface_list 3.9.4
- membership_choice 4.4
- object_renaming_declaration 8.5.1
- parameter_and_result_profile 6.1
- parameter_specification 6.1
- qualified_expression 4.7
- subtype_indication 3.2.2
- type_conversion 4.6
- use_type_clause 8.4

subunit 10.1.3
- compilation_unit 10.1.1

target_name 5.2.1
- name 4.1

task_body 9.1
- proper_body 3.11

task_body_stub 10.1.3
- body_stub 10.1.3

task_definition 9.1
- single_task_declaration 9.1
- task_type_declaration 9.1

task_item 9.1
- task_definition 9.1

task_type_declaration 9.1
- full_type_declaration 3.2.1

term 4.4
- simple_expression 4.4

terminate_alternative 9.7.1
- select_alternative 9.7.1

timed_entry_call 9.7.2
- select_statement 9.7

triggering_alternative 9.7.4
- asynchronous_select 9.7.4

triggering_statement 9.7.4
- triggering_alternative 9.7.4

type_conversion 4.6
- name 4.1

type_declaration 3.2.1
- basic_declaration 3.1

type_definition 3.2.1
- full_type_declaration 3.2.1

unary_adding_operator 4.5
- simple_expression 4.4

unconstrained_array_definition 3.6
- array_type_definition 3.6

underline ...
- based_numeral 2.4.2
- numeral 2.4.1

unknown_discriminant_part 3.7
- discriminant_part 3.7

use_clause 8.4
- basic_declarative_item 3.11
- context_item 10.1.2
- generic_formal_part 12.1

use_package_clause 8.4
- use_clause 8.4

use_type_clause 8.4
- use_clause 8.4

value_sequence 4.5.10
- reduction_attribute_reference 4.5.10

variant 3.8.1
- variant_part 3.8.1

variant_part 3.8.1
- component_list 3.8

with_clause 10.1.2
- context_item 10.1.2

Annex Q
(informative)
Language-Defined Entities

This annex lists the language-defined entities of the language. A list of language-defined library units can be found in Annex A, “Predefined Language Environment”. 1/2

Q.1 Language-Defined Packages

This subclause lists all language-defined packages. 1/3

Ada A.2(2/5)
Address_To_Access_Conversions
  child of System 13.7.2(2/5)
Arithmetic
  child of Ada.Calendar 9.6.1(8/5)
ASCII
  in Standard A.1(36.3/2)
Assertions
  child of Ada 11.4.2(12/5)
Asynchronous_Task_Control
  child of Ada D.11(3/5)
Atomic_Operations
  child of System C.6.1(3/5)
Big_Integers
  child of Ada.Numerics.Big_Numbers A.5.5(3/5), A.5.6(2/5)
Big_Reals
  child of Ada.Numerics.Big_Numbers A.5.7(2/5)
Bounded
  child of Ada.Streams.Storage 13.13.1(26/5)
  child of Ada.Strings A.4.4(3/5)
  child of Ada.Strings.Text_Buffers A.4.12(26/5)
Bounded_Doubly_Linked_Lists
  child of Ada.Containers A.18.20(1/3)
Bounded_Hashed_Maps
  child of Ada.Containers A.18.21(1/3)
Bounded_Hashed_Sets
  child of Ada.Containers A.18.23(1/3)
Bounded_Indefinite_Holders
  child of Ada.Containers A.18.32(1/5)
Bounded_IO
  child of Ada.Text_IO A.10.11(3/5)
Bounded_Multiway_Trees
  child of Ada.Containers A.18.25(1/3)
Bounded_Ordered_Maps
  child of Ada.Containers A.18.22(1/3)
Bounded_Ordered_Sets
  child of Ada.Containers A.18.24(1/3)
Bounded_Priority_Queues
  child of Ada.Containers A.18.31(2/6)
Bounded_Synchronized_Queues
  child of Ada.Containers A.18.29(2/6)
Bounded_Vectors
  child of Ada.Containers A.18.19(1/3)
C
  child of Interfaces B.3(4/5)
Calendar
  child of Ada 9.6(10/5)
Characters
  child of Ada A.3.1(2/5)
COBOL
  child of Interfaces B.4(7/5)
Command_Line
  child of Ada A.15(3/5)
Complex_Arrays
  child of Ada.Numerics G.3.2(53/2)
Complex_Elementary_Functions
  child of Ada.Numerics G.1.2(9/1)
Complex_Text_IO
  child of Ada G.1.3(9.1/2)
Complex_Types
  child of Ada.Numerics G.1.1(25/1)
Complex_IO
  child of Ada.Text_IO G.1.3(3/5)
  child of Ada.Wide_Text_IO G.1.4(1)
  child of Ada.Wide_Wide_Text_IO G.1.5(1/2)
Constants
  child of Ada.Strings.Maps A.4.6(3/5)
Containers
  child of Ada A.18.1(3/5)
Conversions
  child of Ada.Characters A.3.4(2/5)
  child of Ada.Strings.UTF_Encoding A.4.11(15/5)
Decimal
  child of Ada F.2(2/5)
Decimal_Conversions
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(14.1/6)
  in Interfaces.COBOL B.4(31)
Decimal_IO
  in Ada.Text_IO A.10.1(73)
Decimal_Output
  in Ada.Text_IO.Editing F.3.3(11)
Direct_IO
  child of Ada A.8.4(2/5)

Directories
child of Ada A.16(3/5)
Discrete_Random
child of Ada.Numerics A.5.2(17/5)
Dispatching
child of Ada D.2.1(1.2/5)
Dispatching_Domains
child of System.Multiprocessors D.16.1(3/5)
Double_Precision_Complex_Types
in Interfaces.Fortran B.5(10.1/5)
Doubly_Linked_Lists
child of Ada.Containers A.18.3(5/5)
Dynamic_Priorities
child of Ada D.5.1(3/5)
EDF
child of Ada.Dispatching D.2.6(9/5)
child of Ada.Synchronous_Task_Control D.10(5.2/5)
Editing
child of Ada.Text_IO F.3.3(3/5)
child of Ada.Wide_Text_IO F.3.4(1)
child of Ada.Wide_Wide_Text_IO F.3.5(1/2)
Elementary_Functions
child of Ada.Numerics A.5.1(9/1)
Enumeration_IO
in Ada.Text_IO A.10.1(79)
Environment_Variables
child of Ada A.17(3/5)
Exceptions
child of Ada 11.4.1(2/5)
Exchange
child of System.Atomic_Operations C.6.2(5/5)
Execution_Time
child of Ada D.14(3/5)
Finalization
child of Ada 7.6(4/5)
Fixed
child of Ada.Strings A.4.3(5/5)
Fixed_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(14/5)
Fixed_IO
in Ada.Text_IO A.10.1(68)
Float_Random
child of Ada.Numerics A.5.2(5/5)
Float_Text_IO
child of Ada A.10.9(33)
Float_Wide_Text_IO
child of Ada A.11(2/2)
Float_Wide_Wide_Text_IO
child of Ada A.11(3/2)
Float_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5)
Float_IO
in Ada.Text_IO A.10.1(63)
Formatting
child of Ada.Calendar 9.6.1(15/5)
Fortran
child of Interfaces B.5(4/5)
Generic_Complex_Arrays
child of Ada.Numerics G.3.2(2/5)
Generic_Complex_Elementary_Functions
child of Ada.Numerics G.1.2(2/5)
Generic_Complex_Types
child of Ada.Numerics G.1.1(2/5)
Generic_Dispatching_Constructor
child of Ada.Tags 3.9(18.2/5)
Generic_Elementary_Functions
child of Ada.Numerics A.5.1(3/5)
Generic_Bounded_Length
in Ada.Strings.Bounded A.4.4(4)
Generic_Keys
in Ada.Containers.Hashed_Sets A.18.8(50/5)
in Ada.Containers.Ordered_Sets A.18.9(62/5)
Generic_Real_Arrays
child of Ada.Numerics G.3.1(2/5)
Generic_Sorting
in Ada.Containers.Doubly_Linked_Lists A.18.3(47/5)
in Ada.Containers.Vectors A.18.2(75/5)
Group_Budgets
child of Ada.Execution_Time D.14.2(3/5)
Handling
child of Ada.Characters A.3.2(2/5)
child of Ada.Wide_Characters A.3.5(3/5)
child of Ada.Wide_Wide_Characters A.3.6(1/3)
Hashed_Maps
child of Ada.Containers A.18.5(2/5)
Hashed_Sets
child of Ada.Containers A.18.8(2/5)
Hierarchical_File_Names
child of Ada.Directories A.16.1(3/5)
child of Ada.Wide_Directories A.16.2(2/5)
child of Ada.Wide_Wide_Directories A.16.2(3/5)
Indefinite_Doubly_Linked_Lists
child of Ada.Containers A.18.12(2/3)
Indefinite_Hashed_Maps
child of Ada.Containers A.18.13(2/3)
Indefinite_Hashed_Sets
child of Ada.Containers A.18.15(2/3)
Indefinite_Holders
child of Ada.Containers A.18.18(5/5)
Indefinite_Multiway_Trees
child of Ada.Containers A.18.17(2/3)
Indefinite_Ordered_Maps
child of Ada.Containers A.18.14(2/3)
Indefinite_Ordered_Sets
child of Ada.Containers A.18.16(2/3)
Indefinite_Vectors
child of Ada.Containers A.18.11(2/3)
Information
child of Ada.Directories A.16(124/2)
child of Ada.Wide_Directories A.16.2(2/5)
child of Ada.Wide_Wide_Directories A.16.2(3/5)
Integer_Text_IO
child of Ada A.10.8(21)
Integer_Wide_Text_IO
child of Ada A.11(2/2)
Integer_Wide_Wide_Text_IO
child of Ada A.11(3/2)
Integer_Arithmetic
child of System.Atomic_Operations C.6.4(3/5)

Integer_IO
in Ada.Text_IO A.10.1(52)
Interfaces B.2(3/5)
Interrupts
child of Ada C.3.2(2/5)
child of Ada.Execution_Time D.14.3(3/5)
IO_Exceptions
child of Ada A.13(3/5)
Iterator_Interfaces
child of Ada 5.5.1(2/5)
Latin_1
child of Ada.Characters A.3.3(3/5)
List_Iterator_Interfaces
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.3/3), A.18.3(51.7/5)
Locales
child of Ada A.19(3/5)
Machine_Code
child of System 13.8(7)
Map_Iterator_Interfaces
in Ada.Containers.Hashed_Maps A.18.5(6.3/5), A.18.5(37.8/5)
in Ada.Containers.Ordered_Maps A.18.6(7.3/5), A.18.6(51.9/5)
Maps
child of Ada.Strings A.4.2(3/5)
Modular_Arithmetic
child of System.Atomic_Operations C.6.5(3/5)
Modular_IO
in Ada.Text_IO A.10.1(57)
Multiprocessors
child of System D.16(3/5)
Multiway_Trees
child of Ada.Containers A.18.10(7/5)
Names
child of Ada.Interrupts C.3.2(12/5)
Non_Preemptive
child of Ada.Dispatching D.2.4(2.2/5)
Numerics
child of Ada A.5(3/5)
Ordered_Maps
child of Ada.Containers A.18.6(2/5)
Ordered_Sets
child of Ada.Containers A.18.9(2/5)
Pointers
child of Interfaces.C B.3.2(4/5)
Real_Arrays
child of Ada.Numerics G.3.1(31/2)
Real_Time
child of Ada D.8(3/5)
Round_Robin
child of Ada.Dispatching D.2.5(4/5)
RPC
child of System E.5(3/5)
Sequential_IO
child of Ada A.8.1(2/5)
Set_Iterator_Interfaces
in Ada.Containers.Hashed_Sets A.18.8(6.3/3), A.18.8(59.7/5)
in Ada.Containers.Ordered_Sets A.18.9(7.3/3), A.18.9(74.7/5)
Signed_Conversions
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5)
Single_Precision_Complex_Types
in Interfaces.Fortran B.5(8)
Stable
in Ada.Containers.Doubly_Linked_Lists A.18.3(51.1/5)
in Ada.Containers.Hashed_Maps A.18.5(37.2/5)
in Ada.Containers.Hashed_Sets A.18.8(59.1/5)
in Ada.Containers.Multiway_Trees A.18.10(70.1/5)
in Ada.Containers.Ordered_Maps A.18.6(51.3/5)
in Ada.Containers.Ordered_Sets A.18.9(74.1/5)
in Ada.Containers.Vectors A.18.2(79.1/5)
Standard A.1(4/5)
Storage
child of Ada.Streams 13.13.1(12/5)
Storage_Elements
child of System 13.7.1(2/5)
Storage_IO
child of Ada A.9(3/5)
Storage_Pools
child of System 13.11(5/5)
Stream_IO
child of Ada.Streams A.12.1(3/5)
Streams
child of Ada 13.13.1(2/5)
Strings
child of Ada A.4.1(3/5)
child of Ada.Strings.UTF_Encoding A.4.11(22/5)
child of Interfaces.C B.3.1(3/5)
Subpools
child of System.Storage_Pools 13.11.4(3/5)
Synchronized_Queue_Interfaces
child of Ada.Containers A.18.27(3/5)
Synchronous_Barriers
child of Ada D.10.1(3/5)
Synchronous_Task_Control
child of Ada D.10(3/5)
System 13.7(3/5)
Tags
child of Ada 3.9(6/5)
Task_Attributes
child of Ada C.7.2(2/5)
Task_Identification
child of Ada C.7.1(2/5)
Task_Termination
child of Ada C.7.3(2/5)
Test_And_Set
child of System.Atomic_Operations C.6.3(3/5)
Text_Streams
child of Ada.Text_IO A.12.2(3/5)
child of Ada.Wide_Text_IO A.12.3(3/5)
child of Ada.Wide_Wide_Text_IO A.12.4(3/5)
Text_Buffers
child of Ada.Strings A.4.12(3/5)

Text_IO
  child of Ada A.10.1(2/5)
Time_Zones
  child of Ada.Calendar 9.6.1(2/5)
Timers
  child of Ada.Execution_Time D.14.1(3/5)
Timing_Events
  child of Ada.Real_Time D.15(3/5)
Tree_Iterator_Interfaces
  in Ada.Containers.Multiway_Trees A.18.10(13/3), A.18.10(70.7/5)
Unbounded
  child of Ada.Streams.Storage 13.13.1(18/5)
  child of Ada.Strings A.4.5(3/5)
  child of Ada.Strings.Text_Buffers A.4.12(18/5)
Unbounded_IO
  child of Ada.Text_IO A.10.12(3/5)
Unbounded_Priority_Queues
  child of Ada.Containers A.18.30(2/5)
Unbounded_Synchronized_Queues
  child of Ada.Containers A.18.28(2/5)
Unsigned_Conversions
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(13/5)
UTF_Encoding
  child of Ada.Strings A.4.11(3/5)
Vector_Iterator_Interfaces
  in Ada.Containers.Vectors A.18.2(11.3/5), A.18.2(79.7/5)
Vectors
  child of Ada.Containers A.18.2(6/5)
Wide_Bounded
  child of Ada.Strings A.4.7(1/3)
Wide_Constants
  child of Ada.Strings.Wide_Maps A.4.7(1/3), A.4.8(28/2)
Wide_Fixed
  child of Ada.Strings A.4.7(1/3)
Wide_Maps
  child of Ada.Strings A.4.7(3/5)
Wide_Text_IO
  child of Ada A.11(2/2)
Wide_Unbounded
  child of Ada.Strings A.4.7(1/3)
Wide_Bounded_IO
  child of Ada.Wide_Text_IO A.11(4/3)
Wide_Characters
  child of Ada A.3.1(4/5)
Wide_Command_Line
  child of Ada A.15.1(2/5)
Wide_Directories
  child of Ada A.16.2(2/5)
Wide_Environment_Variables
  child of Ada A.17.1(2/5)
Wide_File_Names
  in Ada.Direct_IO A.8.4(18.1/5)
  in Ada.Sequential_IO A.8.1(15.1/5)
  in Ada.Streams.Stream_IO A.12.1(26.1/5)
  in Ada.Text_IO A.10.1(85.1/5)
Wide_Strings
  child of Ada.Strings.UTF_Encoding A.4.11(30/5)
Wide_Unbounded_IO
  child of Ada.Wide_Text_IO A.11(5/3)
Wide_Wide_Constants
  child of Ada.Strings.Wide_Wide_Maps A.4.8(1/3)
Wide_Wide_Text_IO
  child of Ada A.11(3/2)
Wide_Wide_Bounded
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Bounded_IO
  child of Ada.Wide_Wide_Text_IO A.11(4/3)
Wide_Wide_Characters
  child of Ada A.3.1(6/5)
Wide_Wide_Command_Line
  child of Ada A.15.1(3/5)
Wide_Wide_Directories
  child of Ada A.16.2(3/5)
Wide_Wide_Environment_Variables
  child of Ada A.17.1(3/5)
Wide_Wide_File_Names
  in Ada.Direct_IO A.8.4(18.8/5)
  in Ada.Sequential_IO A.8.1(15.8/5)
  in Ada.Streams.Stream_IO A.12.1(26.8/5)
  in Ada.Text_IO A.10.1(85.8/5)
Wide_Wide_Fixed
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Maps
  child of Ada.Strings A.4.8(3/5)
Wide_Wide_Strings
  child of Ada.Strings.UTF_Encoding A.4.11(38/5)
Wide_Wide_Unbounded
  child of Ada.Strings A.4.8(1/3)
Wide_Wide_Unbounded_IO
  child of Ada.Wide_Wide_Text_IO A.11(5/3)

Q.2 Language-Defined Types and Subtypes

1/3 This subclause lists all language-defined types and subtypes.

Address
  in System 13.7(12)
Alignment
  in Ada.Strings A.4.1(6)
Alphanumeric
  in Interfaces.COBOL B.4(16/3)
Any_Priority *subtype of* Integer
  in System 13.7(16)

Attribute_Handle
  in Ada.Task_Attributes C.7.2(3)
Barrier_Limit *subtype of* Positive
  in Ada.Synchronous_Barriers D.10.1(4/3)
Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(3/5)
Big_Natural *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(9/5)
Big_Positive *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(8/5)
Big_Real
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(3/5)
Binary
  in Interfaces.COBOL B.4(10)
Binary_Format
  in Interfaces.COBOL B.4(24)
Bit_Order
  in System 13.7(15/2)
Boolean
  in Standard A.1(5)
Bounded_String
  in Ada.Strings.Bounded A.4.4(6)
Buffer_Type
  in Ada.Strings.Text_Buffers.Bounded A.4.12(27/5)
  in Ada.Strings.Text_Buffers.Unbounded A.4.12(19/5)
Buffer_Type *subtype of* Storage_Array
  in Ada.Storage_IO A.9(4)
Byte
  in Interfaces.COBOL B.4(29/3)
Byte_Array
  in Interfaces.COBOL B.4(29/3)
C_bool
  in Interfaces.C B.3(13.1/5)
C_float
  in Interfaces.C B.3(15)
Cause_Of_Termination
  in Ada.Task_Termination C.7.3(3/2)
char
  in Interfaces.C B.3(19)
char16_array
  in Interfaces.C B.3(39.5/3)
char16_t
  in Interfaces.C B.3(39.2/2)
char32_array
  in Interfaces.C B.3(39.14/3)
char32_t
  in Interfaces.C B.3(39.11/2)
char_array
  in Interfaces.C B.3(23/3)
char_array_access
  in Interfaces.C.Strings B.3.1(4)
Character
  in Standard A.1(35/3)
Character_Mapping
  in Ada.Strings.Maps A.4.2(20/5)
Character_Mapping_Function
  in Ada.Strings.Maps A.4.2(25)
Character_Range
  in Ada.Strings.Maps A.4.2(6)
Character_Ranges
  in Ada.Strings.Maps A.4.2(7)
Character_Sequence *subtype of* String
  in Ada.Strings.Maps A.4.2(16)
Character_Set
  in Ada.Strings.Maps A.4.2(4/5)
  in Interfaces.Fortran B.5(11)
chars_ptr
  in Interfaces.C.Strings B.3.1(5/5)
chars_ptr_array
  in Interfaces.C.Strings B.3.1(6/2)
Chunk_Index *subtype of* Positive
  in Ada.Iterator_Interfaces 5.5.1(4.2/5)
COBOL_Character
  in Interfaces.COBOL B.4(13)
Complex
  in Ada.Numerics.Generic_Complex_Types G.1.1(3)
  in Interfaces.Fortran B.5(9)
Complex_Matrix
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(4/2)
Complex_Vector
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(4/2)
Constant_Reference_Type
  in Ada.Containers.Doubly_Linked_Lists A.18.3(51.10/6)
  in Ada.Containers.Hashed_Maps A.18.5(37.11/6)
  in Ada.Containers.Hashed_Sets A.18.8(59.10/6)
  in Ada.Containers.Indefinite_Holders A.18.18(16/6)
  in Ada.Containers.Multiway_Trees A.18.10(70.10/6)
  in Ada.Containers.Ordered_Maps A.18.6(51.12/6)
  in Ada.Containers.Ordered_Sets A.18.9(74.10/6)
  in Ada.Containers.Vectors A.18.2(79.10/6)
Controlled
  in Ada.Finalization 7.6(5/5)
Count
  in Ada.Direct_IO A.8.4(4)
  in Ada.Streams.Stream_IO A.12.1(7)
  in Ada.Text_IO A.10.1(5)
Count_Type
  in Ada.Containers A.18.1(5/2)
Country_Code
  in Ada.Locales A.19(4/4)
CPU *subtype of* CPU_Range
  in System.Multiprocessors D.16(4/3)
CPU_Range
  in System.Multiprocessors D.16(4/3)
CPU_Set
  in System.Multiprocessors.Dispatching_Domains D.16.1(9.1/4)
CPU_Time
  in Ada.Execution_Time D.14(4/2)
Cursor
  in Ada.Containers.Doubly_Linked_Lists A.18.3(7/5), A.18.3(51.3/5)
  in Ada.Containers.Hashed_Maps A.18.5(4/5), A.18.5(37.4/5)
  in Ada.Containers.Hashed_Sets A.18.8(4/5), A.18.8(59.3/5)
  in Ada.Containers.Multiway_Trees A.18.10(9/5), A.18.10(70.3/5)
  in Ada.Containers.Ordered_Maps A.18.6(5/5), A.18.6(51.5/5)

in Ada.Containers.Ordered_Sets A.18.9(5/5), A.18.9(74.3/5)
in Ada.Containers.Vectors A.18.2(9/5), A.18.2(79.3/5)
Day_Count
in Ada.Calendar.Arithmetic 9.6.1(10/2)
Day_Duration *subtype of* Duration
in Ada.Calendar 9.6(11/2)
Day_Name
in Ada.Calendar.Formatting 9.6.1(17/2)
Day_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)
Deadline *subtype of* Time
in Ada.Dispatching.EDF D.2.6(9/5)
Decimal_Element
in Interfaces.COBOL B.4(12/3)
Direction
in Ada.Strings A.4.1(6)
Directory_Entry_Type
in Ada.Directories A.16(29/2)
Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(5/3)
Display_Format
in Interfaces.COBOL B.4(22)
double
in Interfaces.C B.3(16)
Double_Complex
in Interfaces.Fortran B.5(10.2/5)
Double_Imaginary *subtype of* Imaginary
in Interfaces.Fortran B.5(10.3/5)
Double_Precision
in Interfaces.Fortran B.5(6)
Duration
in Standard A.1(43)
Encoding_Scheme
in Ada.Strings.UTF_Encoding A.4.11(4/3)
Exception_Id
in Ada.Exceptions 11.4.1(2/5)
Exception_Occurrence
in Ada.Exceptions 11.4.1(3/5)
Exception_Occurrence_Access
in Ada.Exceptions 11.4.1(3/5)
Exit_Status
in Ada.Command_Line A.15(7)
Extended_Index *subtype of* Index_Type'Base
in Ada.Containers.Vectors A.18.2(7/2)
Field *subtype of* Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
in Ada.Text_IO A.10.1(6)
File_Access
in Ada.Text_IO A.10.1(18)
File_Kind
in Ada.Directories A.16(22/2)
File_Mode
in Ada.Direct_IO A.8.4(4)
in Ada.Sequential_IO A.8.1(4)
in Ada.Streams.Stream_IO A.12.1(6)
in Ada.Text_IO A.10.1(4)
File_Size
in Ada.Directories A.16(23/2)
File_Type
in Ada.Direct_IO A.8.4(3)
in Ada.Sequential_IO A.8.1(3)
in Ada.Streams.Stream_IO A.12.1(5/5)
in Ada.Text_IO A.10.1(3)
Filter_Type
in Ada.Directories A.16(30/2)
Float
in Standard A.1(21)
Floating
in Interfaces.COBOL B.4(9)
Fortran_Character
in Interfaces.Fortran B.5(12/3)
Fortran_Integer
in Interfaces.Fortran B.5(5)
Forward_Iterator
in Ada.Iterator_Interfaces 5.5.1(3/3)
Generator
in Ada.Numerics.Discrete_Random A.5.2(19)
in Ada.Numerics.Float_Random A.5.2(7)
Group_Budget
in Ada.Execution_Time.Group_Budgets D.14.2(4/3)
Group_Budget_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(5/5)
Hash_Type
in Ada.Containers A.18.1(4/2)
Holder
in Ada.Containers.Indefinite_Holders A.18.18(6/5)
Hour_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
Imaginary
in Ada.Numerics.Generic_Complex_Types G.1.1(4/5)
Imaginary *subtype of* Imaginary
in Interfaces.Fortran B.5(10)
int
in Interfaces.C B.3(7)
Integer
in Standard A.1(12)
Integer_Address
in System.Storage_Elements 13.7.1(10/3)
Interrupt_Id
in Ada.Interrupts C.3.2(2/5)
Interrupt_Priority *subtype of* Any_Priority
in System 13.7(16)
ISO_646 *subtype of* Character
in Ada.Characters.Handling A.3.2(9)
Language_Code
in Ada.Locales A.19(4/4)
Leap_Seconds_Count *subtype of* Integer
in Ada.Calendar.Arithmetic 9.6.1(11/2)
Length_Range *subtype of* Natural
in Ada.Strings.Bounded A.4.4(8)
Limited_Controlled
in Ada.Finalization 7.6(7/5)
List
in Ada.Containers.Doubly_Linked_Lists A.18.3(6/5), A.18.3(51.2/5)
Logical
in Interfaces.Fortran B.5(7)

long
in Interfaces.C B.3(7)
Long_Binary
in Interfaces.COBOL B.4(10)
long_double
in Interfaces.C B.3(17)
Long_Floating
in Interfaces.COBOL B.4(9)
Map
in Ada.Containers.Hashed_Maps A.18.5(3/5), A.18.5(37.3/5)
in Ada.Containers.Ordered_Maps A.18.6(4/5), A.18.6(51.4/5)
Membership
in Ada.Strings A.4.1(6)
Minute_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
Month_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)
Name
in System 13.7(4)
Name_Case_Kind
in Ada.Directories A.16(20.1/3)
Natural *subtype of* Integer
in Standard A.1(13)
Number_Base *subtype of* Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
in Ada.Text_IO A.10.1(6)
Numeric
in Interfaces.COBOL B.4(20/3)
Packed_Decimal
in Interfaces.COBOL B.4(12/3)
Packed_Format
in Interfaces.COBOL B.4(26)
Parallel_Iterator
in Ada.Iterator_Interfaces 5.5.1(4.1/5)
Parallel_Reversible_Iterator
in Ada.Iterator_Interfaces 5.5.1(4.8/5)
Parameterless_Handler
in Ada.Interrupts C.3.2(2/5)
Params_Stream_Type
in System.RPC E.5(6)
Partition_Id
in System.RPC E.5(4)
Picture
in Ada.Text_IO.Editing F.3.3(4)
plain_char
in Interfaces.C B.3(11)
Pointer
in Interfaces.C.Pointers B.3.2(5)
Positive *subtype of* Integer
in Standard A.1(13)
Positive_Count *subtype of* Count
in Ada.Direct_IO A.8.4(4)
in Ada.Streams.Stream_IO A.12.1(7)
in Ada.Text_IO A.10.1(5)
Priority *subtype of* Any_Priority
in System 13.7(16)
ptrdiff_t
in Interfaces.C B.3(12)
Queue
in Ada.Containers.Bounded_Priority_Queues A.18.31(4/6)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(4/6)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(4/3)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(4/3)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(4/3)
Real
in Interfaces.Fortran B.5(6)
Real_Matrix
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
Real_Vector
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
Reference_Type
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.2/6), A.18.3(51.11/6)
in Ada.Containers.Hashed_Maps A.18.5(17.2/6), A.18.5(37.12/6)
in Ada.Containers.Hashed_Sets A.18.8(58.1/6)
in Ada.Containers.Indefinite_Holders A.18.18(17/6)
in Ada.Containers.Multiway_Trees A.18.10(29/6), A.18.10(70.11/6)
in Ada.Containers.Ordered_Maps A.18.6(16.2/6), A.18.6(51.13/6)
in Ada.Containers.Ordered_Sets A.18.9(73.1/6)
in Ada.Containers.Vectors A.18.2(34.2/6), A.18.2(79.11/6)
Relative_Deadline *subtype of* Time_Span
in Ada.Dispatching.EDF D.2.6(9/5)
Reversible_Iterator
in Ada.Iterator_Interfaces 5.5.1(4/3)
Root_Buffer_Type
in Ada.Strings.Text_Buffers A.4.12(6/5)
Root_Storage_Pool
in System.Storage_Pools 13.11(6/5)
Root_Storage_Pool_With_Subpools
in System.Storage_Pools.Subpools 13.11.4(4/5)
Root_Stream_Type
in Ada.Streams 13.13.1(3/2)
Root_Subpool
in System.Storage_Pools.Subpools 13.11.4(5/5)
RPC_Receiver
in System.RPC E.5(11)
Search_Type
in Ada.Directories A.16(31/2)
Second_Duration *subtype of* Day_Duration
in Ada.Calendar.Formatting 9.6.1(20/2)
Second_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
Seconds_Count
in Ada.Real_Time D.8(15)
Set
in Ada.Containers.Hashed_Sets A.18.8(3/5), A.18.8(59.2/5)
in Ada.Containers.Ordered_Sets A.18.9(4/5), A.18.9(74.2/5)
short
in Interfaces.C B.3(7)
signed_char
in Interfaces.C B.3(8)

size_t
in Interfaces.C B.3(13)
State
in Ada.Numerics.Discrete_Random A.5.2(23)
in Ada.Numerics.Float_Random A.5.2(11)
Storage_Array
in System.Storage_Elements 13.7.1(5)
Storage_Count *subtype of* Storage_Offset
in System.Storage_Elements 13.7.1(4)
Storage_Element
in System.Storage_Elements 13.7.1(5)
Storage_Offset
in System.Storage_Elements 13.7.1(3)
Storage_Stream_Type
in Ada.Streams.Storage 13.13.1(13/5)
Stream_Access
in Ada.Streams.Stream_IO A.12.1(4)
in Ada.Text_IO.Text_Streams A.12.2(3/5)
in Ada.Wide_Text_IO.Text_Streams A.12.3(3/5)
in Ada.Wide_Wide_Text_IO.Text_Streams A.12.4(3/5)
Stream_Element
in Ada.Streams 13.13.1(4/1)
Stream_Element_Array
in Ada.Streams 13.13.1(4/1)
Stream_Element_Count *subtype of* Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_Type
in Ada.Streams.Storage.Bounded 13.13.1(27/5)
in Ada.Streams.Storage.Unbounded 13.13.1(19/5)
String
in Standard A.1(37/3)
String_Access
in Ada.Strings.Unbounded A.4.5(7)
Subpool_Handle
in System.Storage_Pools.Subpools 13.11.4(6/3)
Suspension_Object
in Ada.Synchronous_Task_Control D.10(4/5)
Synchronous_Barrier
in Ada.Synchronous_Barriers D.10.1(5/3)
Tag
in Ada.Tags 3.9(6/5)
Tag_Array
in Ada.Tags 3.9(7.3/2)
Task_Array
in Ada.Execution_Time.Group_Budgets D.14.2(6/2)
Task_Id
in Ada.Task_Identification C.7.1(2/5)
Termination_Handler
in Ada.Task_Termination C.7.3(4/2)
Test_And_Set_Flag
in System.Atomic_Operations.Test_And_Set C.6.3(4/5)
Text_Buffer_Count
in Ada.Strings.Text_Buffers A.4.12(4/5)
Time
in Ada.Calendar 9.6(10/5)
in Ada.Real_Time D.8(4)
Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(4/2)
Time_Span
in Ada.Real_Time D.8(5)
Timer
in Ada.Execution_Time.Timers D.14.1(4/2)
Timer_Handler
in Ada.Execution_Time.Timers D.14.1(5/5)
Timing_Event
in Ada.Real_Time.Timing_Events D.15(4/5)
Timing_Event_Handler
in Ada.Real_Time.Timing_Events D.15(4/5)
Tree
in Ada.Containers.Multiway_Trees A.18.10(8/5), A.18.10(70.2/5)
Trim_End
in Ada.Strings A.4.1(6)
Truncation
in Ada.Strings A.4.1(6)
Type_Set
in Ada.Text_IO A.10.1(7)
Unbounded_String
in Ada.Strings.Unbounded A.4.5(4/5)
Uniformly_Distributed *subtype of* Float
in Ada.Numerics.Float_Random A.5.2(8/5)
unsigned
in Interfaces.C B.3(9)
unsigned_char
in Interfaces.C B.3(10)
unsigned_long
in Interfaces.C B.3(9)
unsigned_short
in Interfaces.C B.3(9)
UTF_16_Wide_String *subtype of* Wide_String
in Ada.Strings.UTF_Encoding A.4.11(7/3)
UTF_8_String *subtype of* String
in Ada.Strings.UTF_Encoding A.4.11(6/3)
UTF_String *subtype of* String
in Ada.Strings.UTF_Encoding A.4.11(5/3)
Valid_Big_Integer *subtype of* Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(5/5)
Valid_Big_Real *subtype of* Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(5/5)
Vector
in Ada.Containers.Vectors A.18.2(8/5), A.18.2(79.2/5)
wchar_array
in Interfaces.C B.3(33/3)
wchar_t
in Interfaces.C B.3(30/1)
Wide_Character
in Standard A.1(36.1/6)
Wide_Character_Mapping
in Ada.Strings.Wide_Maps A.4.7(20/5)
Wide_Character_Mapping_Function
in Ada.Strings.Wide_Maps A.4.7(26)
Wide_Character_Range
in Ada.Strings.Wide_Maps A.4.7(6)
Wide_Character_Ranges
in Ada.Strings.Wide_Maps A.4.7(7)
Wide_Character_Sequence *subtype of* Wide_String
in Ada.Strings.Wide_Maps A.4.7(16)

Wide_Character_Set
in Ada.Strings.Wide_Maps A.4.7(4/5)
Wide_String
in Standard A.1(41/3)
Wide_Wide_Character
in Standard A.1(36.2/6)
Wide_Wide_Character_Mapping
in Ada.Strings.Wide_Wide_Maps A.4.8(20/5)
Wide_Wide_Character_Mapping_Function
in Ada.Strings.Wide_Wide_Maps A.4.8(26/2)
Wide_Wide_Character_Range
in Ada.Strings.Wide_Wide_Maps A.4.8(6/2)
Wide_Wide_Character_Ranges
in Ada.Strings.Wide_Wide_Maps A.4.8(7/2)
Wide_Wide_Character_Sequence *subtype of* Wide_Wide_String
in Ada.Strings.Wide_Wide_Maps A.4.8(16/2)
Wide_Wide_Character_Set
in Ada.Strings.Wide_Wide_Maps A.4.8(4/5)
Wide_Wide_String
in Standard A.1(42.1/3)
Year_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)

Q.3 Language-Defined Subprograms

This subclause lists all language-defined subprograms. 1/3

Abort_Task *in* Ada.Task_Identification C.7.1(3/5)
Activation_Is_Complete
in Ada.Task_Identification C.7.1(4/3)
Actual_Quantum
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Ada.Unchecked_Deallocate_Subpool
child of Ada 13.11.5(3/6)
Add
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Add_Task
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Adjust *in* Ada.Finalization 7.6(6/2)
Allocate
in System.Storage_Pools 13.11(7)
in System.Storage_Pools.Subpools 13.11.4(14/3)
Allocate_From_Subpool
in System.Storage_Pools.Subpools 13.11.4(11/5)
Ancestor_Find
in Ada.Containers.Multiway_Trees A.18.10(40.1/5), A.18.10(40/5)
Append
in Ada.Containers.Doubly_Linked_Lists A.18.3(23.1/5), A.18.3(23/5)
in Ada.Containers.Vectors A.18.2(47.1/5), A.18.2(47/5)
in Ada.Strings.Bounded A.4.4(13), A.4.4(14), A.4.4(15), A.4.4(16), A.4.4(17), A.4.4(18), A.4.4(19), A.4.4(20)
in Ada.Strings.Unbounded A.4.5(12), A.4.5(13), A.4.5(14)
Append_Child
in Ada.Containers.Multiway_Trees A.18.10(52/5)
Append_Vector
in Ada.Containers.Vectors A.18.2(46/5)
Arccos
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arccosh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arccot
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arccoth
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arcsin
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arcsinh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Arctan
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(5)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(6)
Arctanh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(7)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Argument
in Ada.Command_Line A.15(5)
in Ada.Numerics.Generic_Complex_Arrays G.3.2(10/2), G.3.2(31/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(10)
Argument_Count *in* Ada.Command_Line A.15(4)
Assert *in* Ada.Assertions 11.4.2(14/2)
Assign
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.5/5), A.18.3(51.8/5)
in Ada.Containers.Hashed_Maps A.18.5(17.7/5), A.18.5(37.9/5)
in Ada.Containers.Hashed_Sets A.18.8(17.4/5), A.18.8(59.8/5)
in Ada.Containers.Indefinite_Holders A.18.18(20/5)

in Ada.Containers.Multiway_Trees A.18.10(32/5), A.18.10(70.8/5)
in Ada.Containers.Ordered_Maps A.18.6(16.7/5), A.18.6(51.10/5)
in Ada.Containers.Ordered_Sets A.18.9(16.4/5), A.18.9(74.8/5)
in Ada.Containers.Vectors A.18.2(34.7/5), A.18.2(79.8/5)
Assign_Task
in System.Multiprocessors.Dispatching_Domains D.16.1(11/3)
Atomic_Add
in System.Atomic_Operations.Integer_Arithmetic C.6.4(4/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(4/5)
Atomic_Clear
in System.Atomic_Operations.Test_And_Set C.6.3(6/5)
Atomic_Compare_And_Exchange
in System.Atomic_Operations.Exchange C.6.2(7/5)
Atomic_Exchange
in System.Atomic_Operations.Exchange C.6.2(6/5)
Atomic_Fetch_And_Add
in System.Atomic_Operations.Integer_Arithmetic C.6.4(6/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(6/5)
Atomic_Fetch_And_Subtract
in System.Atomic_Operations.Integer_Arithmetic C.6.4(7/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(7/5)
Atomic_Subtract
in System.Atomic_Operations.Integer_Arithmetic C.6.4(5/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(5/5)
Atomic_Test_And_Set
in System.Atomic_Operations.Test_And_Set C.6.3(5/5)
Attach_Handler *in* Ada.Interrupts C.3.2(7)
Base_Name *in* Ada.Directories A.16(19/5)
Blank_When_Zero
in Ada.Text_IO.Editing F.3.3(7)
Bounded_Slice *in* Ada.Strings.Bounded A.4.4(28.1/2), A.4.4(28.2/2)
Budget_Has_Expired
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Budget_Remaining
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Cancel_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Real_Time.Timing_Events D.15(5/2)
Capacity
in Ada.Containers.Hashed_Maps A.18.5(8/5)
in Ada.Containers.Hashed_Sets A.18.8(10/5)
in Ada.Containers.Vectors A.18.2(19/5)
Ceiling
in Ada.Containers.Ordered_Maps A.18.6(41/5)
in Ada.Containers.Ordered_Sets A.18.9(51/2), A.18.9(71/5)
Character_Set_Version
in Ada.Wide_Characters.Handling A.3.5(4/3)
Child_Count
in Ada.Containers.Multiway_Trees A.18.10(46.1/5), A.18.10(46/5)
Child_Depth
in Ada.Containers.Multiway_Trees A.18.10(47.1/5), A.18.10(47/5)
Chunk_Count
in Ada.Iterator_Interfaces 5.5.1(4.5/5)
Clear
in Ada.Containers.Doubly_Linked_Lists A.18.3(13/5)
in Ada.Containers.Hashed_Maps A.18.5(12/5)
in Ada.Containers.Hashed_Sets A.18.8(14/5)
in Ada.Containers.Indefinite_Holders A.18.18(11/5)
in Ada.Containers.Multiway_Trees A.18.10(23/5)
in Ada.Containers.Ordered_Maps A.18.6(11/5)
in Ada.Containers.Ordered_Sets A.18.9(13/5)
in Ada.Containers.Vectors A.18.2(24/5)
in Ada.Environment_Variables A.17(7/2)
in Ada.Streams.Storage 13.13.1(15/5)
in Ada.Streams.Storage.Bounded 13.13.1(31/5)
in Ada.Streams.Storage.Unbounded 13.13.1(23/5)
Clock
in Ada.Calendar 9.6(12)
in Ada.Execution_Time D.14(5/2)
in Ada.Execution_Time.Interrupts D.14.3(3/5)
in Ada.Real_Time D.8(6)
Clock_For_Interrupts
in Ada.Execution_Time D.14(9.3/3)
Close
in Ada.Direct_IO A.8.4(8)
in Ada.Sequential_IO A.8.1(8)
in Ada.Streams.Stream_IO A.12.1(10)
in Ada.Text_IO A.10.1(11)
Col *in* Ada.Text_IO A.10.1(37/5)
Command_Name *in* Ada.Command_Line A.15(6)
Compose
in Ada.Directories A.16(20/5)
in Ada.Directories.Hierarchical_File_Names A.16.1(14/3)
Compose_From_Cartesian
in Ada.Numerics.Generic_Complex_Arrays G.3.2(9/2), G.3.2(29/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(8)
Compose_From_Polar
in Ada.Numerics.Generic_Complex_Arrays G.3.2(11/2), G.3.2(32/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(11)
Conjugate
in Ada.Numerics.Generic_Complex_Arrays G.3.2(13/2), G.3.2(34/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(12), G.1.1(15)
Constant_Reference
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.3/5)
in Ada.Containers.Hashed_Maps A.18.5(17.3/5), A.18.5(17.5/5)
in Ada.Containers.Hashed_Sets A.18.8(17.3/5), A.18.8(58.3/5)
in Ada.Containers.Indefinite_Holders A.18.18(18/5)
in Ada.Containers.Multiway_Trees A.18.10(30/5)
in Ada.Containers.Ordered_Maps A.18.6(16.3/5), A.18.6(16.5/5)
in Ada.Containers.Ordered_Sets A.18.9(16.3/5), A.18.9(73.3/5)

in Ada.Containers.Vectors A.18.2(34.3/5), A.18.2(34.5/5)
Containing_Directory
in Ada.Directories A.16(17/5)
in Ada.Directories.Hierarchical_File_Names A.16.1(11/3)
Contains
in Ada.Containers.Doubly_Linked_Lists A.18.3(43/2)
in Ada.Containers.Hashed_Maps A.18.5(32/2)
in Ada.Containers.Hashed_Sets A.18.8(44/2), A.18.8(57/2)
in Ada.Containers.Multiway_Trees A.18.10(41/3)
in Ada.Containers.Ordered_Maps A.18.6(42/2)
in Ada.Containers.Ordered_Sets A.18.9(52/2), A.18.9(72/2)
in Ada.Containers.Vectors A.18.2(71/2)
Continue
in Ada.Asynchronous_Task_Control D.11(3/5)
Convert
in Ada.Strings.UTF_Encoding.Conversions A.4.11(16/3), A.4.11(17/3), A.4.11(18/3), A.4.11(19/3), A.4.11(20/3)
Copy
in Ada.Containers.Bounded_Doubly_Linked_Lists A.18.20(10/5)
in Ada.Containers.Bounded_Hashed_Maps A.18.21(13/5)
in Ada.Containers.Bounded_Hashed_Sets A.18.23(13/5)
in Ada.Containers.Bounded_Ordered_Maps A.18.22(10/5)
in Ada.Containers.Bounded_Ordered_Sets A.18.24(10/5)
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.6/5), A.18.3(51.9/5)
in Ada.Containers.Hashed_Maps A.18.5(17.8/5), A.18.5(37.10/5)
in Ada.Containers.Hashed_Sets A.18.8(17.5/5), A.18.8(59.9/5)
in Ada.Containers.Indefinite_Holders A.18.18(21/5)
in Ada.Containers.Multiway_Trees A.18.10(33/5), A.18.10(70.9/5)
in Ada.Containers.Ordered_Maps A.18.6(16.8/5), A.18.6(51.11/5)
in Ada.Containers.Ordered_Sets A.18.9(16.5/5), A.18.9(74.9/5)
in Ada.Containers.Vectors A.18.2(34.8/5), A.18.2(79.9/5)
Copy_Array *in* Interfaces.C.Pointers B.3.2(15)
Copy_File *in* Ada.Directories A.16(13/2)
Copy_Local_Subtree
in Ada.Containers.Multiway_Trees A.18.10(54.1/5)
Copy_Subtree
in Ada.Containers.Multiway_Trees A.18.10(54.2/5), A.18.10(54/5)
Copy_Terminated_Array
in Interfaces.C.Pointers B.3.2(14)
Cos
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Cosh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Cot
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Coth
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Count
in Ada.Strings.Bounded A.4.4(48), A.4.4(49/6), A.4.4(50)
in Ada.Strings.Fixed A.4.3(13), A.4.3(14/6), A.4.3(15)
in Ada.Strings.Unbounded A.4.5(43), A.4.5(44/6), A.4.5(45)
Country *in* Ada.Locales A.19(6/3)
Create
in Ada.Direct_IO A.8.4(6), A.8.4(18.10/5), A.8.4(18.3/5)
in Ada.Sequential_IO A.8.1(6), A.8.1(15.10/5), A.8.1(15.3/5)
in Ada.Streams.Stream_IO A.12.1(8), A.12.1(26.10/5), A.12.1(26.3/5)
in Ada.Text_IO A.10.1(9), A.10.1(85.10/5), A.10.1(85.3/5)
in System.Multiprocessors.Dispatching_Domains D.16.1(7/4), D.16.1(9.2/4)
Create_Directory *in* Ada.Directories A.16(7/2)
Create_Path *in* Ada.Directories A.16(9/2)
Create_Subpool
in System.Storage_Pools.Subpools 13.11.4(7/3)
Current_Directory *in* Ada.Directories A.16(5/2)
Current_Error *in* Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Interrupts C.3.2(6)
in Ada.Real_Time.Timing_Events D.15(5/2)
Current_Indent
in Ada.Strings.Text_Buffers A.4.12(14/5)
Current_Input *in* Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Output *in* Ada.Text_IO A.10.1(17), A.10.1(20)
Current_State
in Ada.Synchronous_Task_Control D.10(4/5)
Current_Task
in Ada.Task_Identification C.7.1(3/5)
Current_Task_Fallback_Handler
in Ada.Task_Termination C.7.3(5/2)
Current_Use
in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
Day
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(23/2)
Day_Of_Week
in Ada.Calendar.Formatting 9.6.1(18.1/6), 9.6.1(18/2)
Deallocate
in System.Storage_Pools 13.11(8)
in System.Storage_Pools.Subpools 13.11.4(15/3)
Deallocate_Subpool
in System.Storage_Pools.Subpools 13.11.4(12/3)

Decode
  in Ada.Strings.UTF_Encoding.Strings A.4.11(26/3), A.4.11(27/3), A.4.11(28.1/6), A.4.11(28.2/6), A.4.11(28.3/6), A.4.11(28/3)
  in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(34/3), A.4.11(35/3), A.4.11(36.1/6), A.4.11(36.2/6), A.4.11(36.3/6), A.4.11(36/3)
  in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(42/3), A.4.11(43/3), A.4.11(44/3), A.4.11(82.1/6), A.4.11(82.3/6), A.4.11(82.5/6), A.4.11(94.1/6), A.4.11(94.3/6), A.4.11(94.5/6)
Decrease_Indent
  in Ada.Strings.Text_Buffers A.4.12(16/5)
Decrement *in* Interfaces.C.Pointers B.3.2(11/3)
Default_Modulus
  in Ada.Containers.Bounded_Hashed_Maps A.18.21(10/3)
  in Ada.Containers.Bounded_Hashed_Sets A.18.23(10/3)
Default_Subpool_for_Pool
  in System.Storage_Pools.Subpools 13.11.4(13/3)
Delay_Until_And_Set_CPU
  in System.Multiprocessors.Dispatching_Domains D.16.1(14/3)
Delay_Until_And_Set_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
Delete
  in Ada.Containers.Doubly_Linked_Lists A.18.3(24/5)
  in Ada.Containers.Hashed_Maps A.18.5(25/5), A.18.5(26/5)
  in Ada.Containers.Hashed_Sets A.18.8(24/5), A.18.8(25/5), A.18.8(55/5)
  in Ada.Containers.Ordered_Maps A.18.6(24/5), A.18.6(25/5)
  in Ada.Containers.Ordered_Sets A.18.9(23/5), A.18.9(24/5), A.18.9(68/5)
  in Ada.Containers.Vectors A.18.2(50/5), A.18.2(51/5)
  in Ada.Direct_IO A.8.4(8)
  in Ada.Sequential_IO A.8.1(8)
  in Ada.Streams.Stream_IO A.12.1(10)
  in Ada.Strings.Bounded A.4.4(64), A.4.4(65)
  in Ada.Strings.Fixed A.4.3(29), A.4.3(30)
  in Ada.Strings.Unbounded A.4.5(59), A.4.5(60)
  in Ada.Text_IO A.10.1(11)
Delete_Children
  in Ada.Containers.Multiway_Trees A.18.10(53/5)
Delete_Directory *in* Ada.Directories A.16(8/2)
Delete_File *in* Ada.Directories A.16(11/2)
Delete_First
  in Ada.Containers.Doubly_Linked_Lists A.18.3(25/5)
  in Ada.Containers.Ordered_Maps A.18.6(26/5)
  in Ada.Containers.Ordered_Sets A.18.9(25/5)
  in Ada.Containers.Vectors A.18.2(52/5)
Delete_Last
  in Ada.Containers.Doubly_Linked_Lists A.18.3(26/5)
  in Ada.Containers.Ordered_Maps A.18.6(27/5)
  in Ada.Containers.Ordered_Sets A.18.9(26/5)
  in Ada.Containers.Vectors A.18.2(53/5)
Delete_Leaf
  in Ada.Containers.Multiway_Trees A.18.10(35/5)
Delete_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(36/5)
Delete_Tree *in* Ada.Directories A.16(10/2)
Denominator
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(8/5)
Depth
  in Ada.Containers.Multiway_Trees A.18.10(19.1/5), A.18.10(19/5)
Dequeue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(6/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
Dequeue_Only_High_Priority
  in Ada.Containers.Bounded_Priority_Queues A.18.31(6/3)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(6/3)
Dereference_Error
  in Interfaces.C.Strings B.3.1(12)
Descendant_Tag *in* Ada.Tags 3.9(7.1/2)
Detach_Handler *in* Ada.Interrupts C.3.2(9)
Determinant
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Difference
  in Ada.Calendar.Arithmetic 9.6.1(12/2)
  in Ada.Containers.Hashed_Sets A.18.8(32/5), A.18.8(33/5)
  in Ada.Containers.Ordered_Sets A.18.9(33/5), A.18.9(34/5)
Divide *in* Ada.Decimal F.2(6/3)
Do_APC *in* System.RPC E.5(10)
Do_RPC *in* System.RPC E.5(9)
Eigensystem
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(49/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(27/2)
Eigenvalues
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(48/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(26/2)
Element
  in Ada.Containers.Doubly_Linked_Lists A.18.3(14.1/5), A.18.3(14/5)
  in Ada.Containers.Hashed_Maps A.18.5(14.1/5), A.18.5(14/5), A.18.5(31/2)
  in Ada.Containers.Hashed_Sets A.18.8(15.1/5), A.18.8(15/5), A.18.8(52/2)
  in Ada.Containers.Indefinite_Holders A.18.18(12/5)
  in Ada.Containers.Multiway_Trees A.18.10(24.1/5), A.18.10(24/5)
  in Ada.Containers.Ordered_Maps A.18.6(13.1/5), A.18.6(13/5), A.18.6(39/2)
  in Ada.Containers.Ordered_Sets A.18.9(14.1/5), A.18.9(14/5), A.18.9(65/2)
  in Ada.Containers.Vectors A.18.2(27/5), A.18.2(28.1/5), A.18.2(28/5)
  in Ada.Strings.Bounded A.4.4(26)
  in Ada.Strings.Unbounded A.4.5(20)
Element_Count
  in Ada.Streams.Storage 13.13.1(14/5)
  in Ada.Streams.Storage.Bounded 13.13.1(30/5)

in Ada.Streams.Storage.Unbounded 13.13.1(22/5)
Empty
in Ada.Containers.Doubly_Linked_Lists A.18.3(10.3/5)
in Ada.Containers.Hashed_Maps A.18.5(7.3/5)
in Ada.Containers.Hashed_Sets A.18.8(8.2/5)
in Ada.Containers.Indefinite_Holders A.18.18(8.2/5)
in Ada.Containers.Multiway_Trees A.18.10(15.3/5)
in Ada.Containers.Ordered_Maps A.18.6(8.3/5)
in Ada.Containers.Ordered_Sets A.18.9(9.2/5)
in Ada.Containers.Vectors A.18.2(12.4/5)
Encode
in Ada.Strings.UTF_Encoding.Strings A.4.11(23/3), A.4.11(24/3), A.4.11(25/3)
in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(31/3), A.4.11(32/3), A.4.11(33/3)
in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(39/3), A.4.11(40/3), A.4.11(41/3)
Encoding *in* Ada.Strings.UTF_Encoding A.4.11(13/3)
End_Of_File
in Ada.Direct_IO A.8.4(16)
in Ada.Sequential_IO A.8.1(13)
in Ada.Streams.Stream_IO A.12.1(12)
in Ada.Text_IO A.10.1(34/5)
End_Of_Line *in* Ada.Text_IO A.10.1(30/5)
End_Of_Page *in* Ada.Text_IO A.10.1(33/5)
End_Search *in* Ada.Directories A.16(33/2)
Enqueue
in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(5/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
Environment_Task
in Ada.Task_Identification C.7.1(3/5)
Equal_Case_Insensitive
child of Ada.Strings A.4.10(2/5)
child of Ada.Strings.Bounded A.4.10(7/5)
child of Ada.Strings.Fixed A.4.10(5/3)
child of Ada.Strings.Unbounded A.4.10(10/5)
Equal_Element
in Ada.Containers.Indefinite_Holders A.18.18(7.1/5)
in Ada.Containers.Multiway_Trees A.18.10(11.1/5)
Equal_Subtree
in Ada.Containers.Multiway_Trees A.18.10(14/3)
Equivalent_Elements
in Ada.Containers.Hashed_Sets A.18.8(46/5), A.18.8(47/5), A.18.8(48/5)
in Ada.Containers.Ordered_Sets A.18.9(3/2)
Equivalent_Keys
in Ada.Containers.Hashed_Maps A.18.5(34/5), A.18.5(35/5), A.18.5(36/5)
in Ada.Containers.Ordered_Maps A.18.6(3/5)
in Ada.Containers.Ordered_Sets A.18.9(63/2)
Equivalent_Sets
in Ada.Containers.Hashed_Sets A.18.8(8/2)
in Ada.Containers.Ordered_Sets A.18.9(9/2)
Establish_RPC_Receiver *in* System.RPC E.5(12)
Exception_Identity *in* Ada.Exceptions 11.4.1(5/2)
Exception_Information
in Ada.Exceptions 11.4.1(5/2)
Exception_Message *in* Ada.Exceptions 11.4.1(4/3)
Exception_Name *in* Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Exchange_Handler *in* Ada.Interrupts C.3.2(8)
Exclude
in Ada.Containers.Hashed_Maps A.18.5(24/5)
in Ada.Containers.Hashed_Sets A.18.8(23/5), A.18.8(54/5)
in Ada.Containers.Ordered_Maps A.18.6(23/5)
in Ada.Containers.Ordered_Sets A.18.9(22/5), A.18.9(67/5)
Exists
in Ada.Directories A.16(24/2)
in Ada.Environment_Variables A.17(5/2)
Exp
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Expanded_Name *in* Ada.Tags 3.9(7/2)
Extension *in* Ada.Directories A.16(18/5)
External_Tag *in* Ada.Tags 3.9(7/2)
Finalize *in* Ada.Finalization 7.6(6/2), 7.6(8/2)
Find
in Ada.Containers.Doubly_Linked_Lists A.18.3(41/5)
in Ada.Containers.Hashed_Maps A.18.5(30/5)
in Ada.Containers.Hashed_Sets A.18.8(43/5), A.18.8(56/6)
in Ada.Containers.Multiway_Trees A.18.10(38/5)
in Ada.Containers.Ordered_Maps A.18.6(38/5)
in Ada.Containers.Ordered_Sets A.18.9(49/5), A.18.9(69/5)
in Ada.Containers.Vectors A.18.2(68/5)
Find_In_Subtree
in Ada.Containers.Multiway_Trees A.18.10(39.1/5), A.18.10(39/5)
Find_Index *in* Ada.Containers.Vectors A.18.2(67/2)
Find_Token
in Ada.Strings.Bounded A.4.4(50.1/3), A.4.4(51)
in Ada.Strings.Fixed A.4.3(15.1/3), A.4.3(16)
in Ada.Strings.Unbounded A.4.5(45.1/3), A.4.5(46)
First
in Ada.Containers.Doubly_Linked_Lists A.18.3(33/5)
in Ada.Containers.Hashed_Maps A.18.5(27/5)
in Ada.Containers.Hashed_Sets A.18.8(40/5)
in Ada.Containers.Ordered_Maps A.18.6(28/5)
in Ada.Containers.Ordered_Sets A.18.9(41/5)
in Ada.Containers.Vectors A.18.2(58/5)
in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.6/5)
First_Child
in Ada.Containers.Multiway_Trees A.18.10(60.1/5), A.18.10(60/5)
First_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(61.1/5), A.18.10(61/5)
First_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(34/5)
in Ada.Containers.Ordered_Maps A.18.6(29/5)
in Ada.Containers.Ordered_Sets A.18.9(42/5)
in Ada.Containers.Vectors A.18.2(59/5)
First_Index *in* Ada.Containers.Vectors A.18.2(57/5)

First_Key
in Ada.Containers.Ordered_Maps A.18.6(30/5)
Floor
in Ada.Containers.Ordered_Maps A.18.6(40/5)
in Ada.Containers.Ordered_Sets A.18.9(50/2), A.18.9(70/5)
Flush
in Ada.Direct_IO A.8.4(10.1/5)
in Ada.Sequential_IO A.8.1(10.1/5)
in Ada.Streams.Stream_IO A.12.1(25/1)
in Ada.Text_IO A.10.1(21/5)
Form
in Ada.Direct_IO A.8.4(9), A.8.4(18.13/5), A.8.4(18.6/5)
in Ada.Sequential_IO A.8.1(9), A.8.1(15.13/5), A.8.1(15.6/5)
in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.13/5), A.12.1(26.6/5)
in Ada.Text_IO A.10.1(12), A.10.1(85.13/5), A.10.1(85.6/5)
Free
in Ada.Strings.Unbounded A.4.5(7)
in Interfaces.C.Strings B.3.1(11)
From_Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5), A.5.6(13/5)
From_Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5), A.5.7(14.1/6), A.5.7(14/5)
From_Quotient_String
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
From_String
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(15/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(16/5)
From_Universal_Image
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(16/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(17/5), A.5.7(18/5)
Full_Name *in* Ada.Directories A.16(15/5), A.16(39/2)
Generic_Array_Sort
child of Ada.Containers A.18.26(3/5)
Generic_Constrained_Array_Sort
child of Ada.Containers A.18.26(7/5)
Generic_Sort
child of Ada.Containers A.18.26(9.2/5)
Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(20/5)
in Ada.Text_IO A.10.1(41/5), A.10.1(47/5), A.10.1(54/5), A.10.1(55/5), A.10.1(59/5), A.10.1(60/5), A.10.1(65/5), A.10.1(67/5), A.10.1(70/5), A.10.1(72/5), A.10.1(75/5), A.10.1(77/5), A.10.1(81/5), A.10.1(83/5)
in Ada.Text_IO.Complex_IO G.1.3(6), G.1.3(8/5)
Get_CPU
in Ada.Interrupts C.3.2(10.1/3)
in System.Multiprocessors.Dispatching_Domains D.16.1(13/3)
Get_CPU_Set
in System.Multiprocessors.Dispatching_Domains D.16.1(9.3/4)
Get_Deadline *in* Ada.Dispatching.EDF D.2.6(9/5)
Get_Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(10/3)
Get_First_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(8/3)
Get_Immediate *in* Ada.Text_IO A.10.1(44/5), A.10.1(45/5)
Get_Last_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(9/4)
Get_Last_Release_Time
in Ada.Dispatching.EDF D.2.6(9/5)
Get_Line
in Ada.Text_IO A.10.1(49.1/5), A.10.1(49/5)
in Ada.Text_IO.Bounded_IO A.10.11(8/2), A.10.11(9/2), A.10.11(10/2), A.10.11(11/2)
in Ada.Text_IO.Unbounded_IO A.10.12(8/2), A.10.12(9/2), A.10.12(10/2), A.10.12(11/2)
Get_Next_Entry *in* Ada.Directories A.16(35/2)
Get_Priority
in Ada.Dynamic_Priorities D.5.1(5)
Get_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Get_UTF_8
in Ada.Strings.Text_Buffers.Unbounded A.4.12(23/5)
Greatest_Common_Divisor
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(19/5)
Has_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.1/5), A.18.3(9.2/5), A.18.3(51.6/5), A.18.3(69.4/5)
in Ada.Containers.Hashed_Maps A.18.5(6.1/5), A.18.5(6.2/5), A.18.5(37.7/5)
in Ada.Containers.Hashed_Sets A.18.8(6.1/5), A.18.8(6.2/5), A.18.8(59.6/5)
in Ada.Containers.Multiway_Trees A.18.10(12.1/5), A.18.10(12/5), A.18.10(70.6/5), A.18.10(92.1/5)
in Ada.Containers.Ordered_Maps A.18.6(7.1/5), A.18.6(7.2/5), A.18.6(51.8/5)
in Ada.Containers.Ordered_Sets A.18.9(7.1/5), A.18.9(7.2/5), A.18.9(74.6/5)
in Ada.Containers.Vectors A.18.2(11.1/5), A.18.2(11.2/5), A.18.2(79.6/5), A.18.2(97.4/5)
Hash
child of Ada.Strings A.4.9(2/5)
child of Ada.Strings.Bounded A.4.9(7/5)
child of Ada.Strings.Unbounded A.4.9(10/5)
Hash_Case_Insensitive
child of Ada.Strings A.4.9(11.2/5)
child of Ada.Strings.Bounded A.4.9(11.7/5)
child of Ada.Strings.Fixed A.4.9(11.5/3)
child of Ada.Strings.Unbounded A.4.9(11.10/5)
Head
in Ada.Strings.Bounded A.4.4(70), A.4.4(71)
in Ada.Strings.Fixed A.4.3(35), A.4.3(36)
in Ada.Strings.Unbounded A.4.5(65), A.4.5(66)
Hold *in* Ada.Asynchronous_Task_Control D.11(3/5)
Hour *in* Ada.Calendar.Formatting 9.6.1(24/2)
Im
in Ada.Numerics.Generic_Complex_Arrays G.3.2(7/2), G.3.2(27/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(6)
Image
in Ada.Calendar.Formatting 9.6.1(35/2), 9.6.1(37/2)

in Ada.Numerics.Discrete_Random A.5.2(26)
in Ada.Numerics.Float_Random A.5.2(14)
in Ada.Task_Identification C.7.1(3/5)
in Ada.Text_IO.Editing F.3.3(13)
In_Range
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(10/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(12/5)
Include
in Ada.Containers.Hashed_Maps A.18.5(22/5)
in Ada.Containers.Hashed_Sets A.18.8(21/5)
in Ada.Containers.Ordered_Maps A.18.6(21/5)
in Ada.Containers.Ordered_Sets A.18.9(20/5)
Increase_Indent
in Ada.Strings.Text_Buffers A.4.12(15/5)
Increment *in* Interfaces.C.Pointers B.3.2(11/3)
Index
in Ada.Direct_IO A.8.4(15)
in Ada.Streams.Stream_IO A.12.1(23)
in Ada.Strings.Bounded A.4.4(43.1/2), A.4.4(43.2/6), A.4.4(44), A.4.4(45.1/2), A.4.4(45/6), A.4.4(46)
in Ada.Strings.Fixed A.4.3(8.1/2), A.4.3(8.2/6), A.4.3(9), A.4.3(10.1/2), A.4.3(10/6), A.4.3(11)
in Ada.Strings.Unbounded A.4.5(38.1/2), A.4.5(38.2/6), A.4.5(39), A.4.5(40.1/2), A.4.5(40/6), A.4.5(41)
Index_Non_Blank
in Ada.Strings.Bounded A.4.4(46.1/2), A.4.4(47)
in Ada.Strings.Fixed A.4.3(11.1/2), A.4.3(12)
in Ada.Strings.Unbounded A.4.5(41.1/2), A.4.5(42)
Initial_Directory
in Ada.Directories.Hierarchical_File_Names A.16.1(12/3)
Initialize *in* Ada.Finalization 7.6(6/2), 7.6(8/2)
Insert
in Ada.Containers.Doubly_Linked_Lists A.18.3(19/5), A.18.3(20/5), A.18.3(21/5)
in Ada.Containers.Hashed_Maps A.18.5(19/5), A.18.5(20/5), A.18.5(21/5)
in Ada.Containers.Hashed_Sets A.18.8(19/5), A.18.8(20/5)
in Ada.Containers.Ordered_Maps A.18.6(18/5), A.18.6(19/5), A.18.6(20/5)
in Ada.Containers.Ordered_Sets A.18.9(18/5), A.18.9(19/5)
in Ada.Containers.Vectors A.18.2(39/5), A.18.2(40/5), A.18.2(41/5), A.18.2(42/5), A.18.2(43/5)
in Ada.Strings.Bounded A.4.4(60), A.4.4(61)
in Ada.Strings.Fixed A.4.3(25), A.4.3(26)
in Ada.Strings.Unbounded A.4.5(55), A.4.5(56)
Insert_Child
in Ada.Containers.Multiway_Trees A.18.10(48/5), A.18.10(49/5), A.18.10(50/5)
Insert_Space
in Ada.Containers.Vectors A.18.2(48/5), A.18.2(49/5)
Insert_Vector
in Ada.Containers.Vectors A.18.2(36/5), A.18.2(37/5), A.18.2(38/5)
Interface_Ancestor_Tags *in* Ada.Tags 3.9(7.4/2)
Internal_Tag *in* Ada.Tags 3.9(7/2)
Intersection
in Ada.Containers.Hashed_Sets A.18.8(29/5), A.18.8(30/5)
in Ada.Containers.Ordered_Sets A.18.9(30/5), A.18.9(31/5)
Inverse
in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Is_A_Group_Member
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_Abstract *in* Ada.Tags 3.9(7.5/3)
Is_Alphanumeric
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(12/3)
Is_Ancestor_Of
in Ada.Containers.Multiway_Trees A.18.10(21.2/5)
Is_Attached *in* Ada.Interrupts C.3.2(5)
Is_Basic
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(8.1/5)
Is_Callable
in Ada.Task_Identification C.7.1(4/3)
Is_Character
in Ada.Characters.Conversions A.3.4(3/2)
Is_Control
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(5/3)
Is_Current_Directory_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(7/3)
Is_Decimal_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(10/3)
Is_Descendant_At_Same_Level
in Ada.Tags 3.9(7.1/2)
Is_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(9/3)
Is_Empty
in Ada.Containers.Doubly_Linked_Lists A.18.3(12/5)
in Ada.Containers.Hashed_Maps A.18.5(11/5)
in Ada.Containers.Hashed_Sets A.18.8(13/5)
in Ada.Containers.Indefinite_Holders A.18.18(10/5)
in Ada.Containers.Multiway_Trees A.18.10(16/5)
in Ada.Containers.Ordered_Maps A.18.6(10/5)
in Ada.Containers.Ordered_Sets A.18.9(12/5)
in Ada.Containers.Vectors A.18.2(23/5)
Is_Full_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(8/3)
Is_Graphic
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(19/3)
Is_Held
in Ada.Asynchronous_Task_Control D.11(3/5)
Is_Hexadecimal_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(11/3)
Is_In
in Ada.Strings.Maps A.4.2(13)
in Ada.Strings.Wide_Maps A.4.7(13)
in Ada.Strings.Wide_Wide_Maps A.4.8(13/2)
Is_ISO_646 *in* Ada.Characters.Handling A.3.2(10)
Is_Leaf
in Ada.Containers.Multiway_Trees A.18.10(21.1/5), A.18.10(21/5)
Is_Letter

in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(6/3)
Is_Line_Terminator
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(14/3)
Is_Lock_Free
in System.Atomic_Operations.Exchange C.6.2(8/5)
in System.Atomic_Operations.Integer_Arithmetic C.6.4(8/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(8/5)
in System.Atomic_Operations.Test_And_Set C.6.3(7/5)
Is_Lower
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(7/3)
Is_Mark
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(15/3)
Is_Member
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_NFKC
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(18.1/5)
Is_Nul_Terminated *in* Interfaces.C B.3(24), B.3(35), B.3(39.16/2), B.3(39.7/2)
Is_Open
in Ada.Direct_IO A.8.4(10)
in Ada.Sequential_IO A.8.1(10)
in Ada.Streams.Stream_IO A.12.1(12)
in Ada.Text_IO A.10.1(13)
Is_Other_Format
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(16/3)
Is_Parent_Directory_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(6/3)
Is_Punctuation_Connector
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(17/3)
Is_Relative_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(9/3)
Is_Reserved *in* Ada.Interrupts C.3.2(4)
Is_Root
in Ada.Containers.Multiway_Trees A.18.10(20.1/5), A.18.10(20/5)
Is_Root_Directory_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(5/3)
Is_Round_Robin
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Is_Simple_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(4/3)
Is_Sorted
in Ada.Containers.Doubly_Linked_Lists A.18.3(48/2)
in Ada.Containers.Vectors A.18.2(76/2)
Is_Space
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(18/3)
Is_Special
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(13/3)
Is_Split *in* Ada.Iterator_Interfaces 5.5.1(4.3/5)
Is_String
in Ada.Characters.Conversions A.3.4(3/2)
Is_Subset
in Ada.Containers.Hashed_Sets A.18.8(39/2)
in Ada.Containers.Ordered_Sets A.18.9(40/2)
in Ada.Strings.Maps A.4.2(14)
in Ada.Strings.Wide_Maps A.4.7(14)
in Ada.Strings.Wide_Wide_Maps A.4.8(14/2)
Is_Terminated
in Ada.Task_Identification C.7.1(4/3)
Is_Upper
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(8/3)
Is_Valid
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(4/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(4/5)
Is_Wide_Character
in Ada.Characters.Conversions A.3.4(3/2)
Is_Wide_String
in Ada.Characters.Conversions A.3.4(3/2)
Iterate
in Ada.Containers.Doubly_Linked_Lists A.18.3(45/5), A.18.3(46.1/5), A.18.3(46.2/5)
in Ada.Containers.Hashed_Maps A.18.5(37/5)
in Ada.Containers.Hashed_Sets A.18.8(49/5)
in Ada.Containers.Multiway_Trees A.18.10(42/5), A.18.10(44/5)
in Ada.Containers.Ordered_Maps A.18.6(50/5), A.18.6(51.1/5), A.18.6(51.2/5)
in Ada.Containers.Ordered_Sets A.18.9(60/5)
in Ada.Containers.Vectors A.18.2(73/5), A.18.2(74.1/5), A.18.2(74.2/5)
in Ada.Environment_Variables A.17(8/5)
Iterate_Children
in Ada.Containers.Multiway_Trees A.18.10(68.1/5), A.18.10(68/5), A.18.10(70/5)
Iterate_Subtree
in Ada.Containers.Multiway_Trees A.18.10(43.1/5), A.18.10(43/5), A.18.10(45.1/5), A.18.10(45/5)
Key
in Ada.Containers.Hashed_Maps A.18.5(13.1/5), A.18.5(13/5)
in Ada.Containers.Hashed_Sets A.18.8(51.1/5), A.18.8(51/5)
in Ada.Containers.Ordered_Maps A.18.6(12.1/5), A.18.6(12/5)
in Ada.Containers.Ordered_Sets A.18.9(64.1/5), A.18.9(64/5)
Kind *in* Ada.Directories A.16(25/2), A.16(40/2)
Language *in* Ada.Locales A.19(6/3)
Last
in Ada.Containers.Doubly_Linked_Lists A.18.3(35/5)
in Ada.Containers.Ordered_Maps A.18.6(31/5)
in Ada.Containers.Ordered_Sets A.18.9(43/5)
in Ada.Containers.Vectors A.18.2(61/5)
in Ada.Iterator_Interfaces 5.5.1(4/3)
Last_Child
in Ada.Containers.Multiway_Trees A.18.10(62.1/5), A.18.10(62/5)

Last_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(63.1/5), A.18.10(63/5)
Last_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(36/5)
in Ada.Containers.Ordered_Maps A.18.6(32/5)
in Ada.Containers.Ordered_Sets A.18.9(44/5)
in Ada.Containers.Vectors A.18.2(62/5)
Last_Index *in* Ada.Containers.Vectors A.18.2(60/5)
Last_Key
in Ada.Containers.Ordered_Maps A.18.6(33/5)
Length
in Ada.Containers.Doubly_Linked_Lists A.18.3(11/5)
in Ada.Containers.Hashed_Maps A.18.5(10/5)
in Ada.Containers.Hashed_Sets A.18.8(12/5)
in Ada.Containers.Ordered_Maps A.18.6(9/5)
in Ada.Containers.Ordered_Sets A.18.9(11/5)
in Ada.Containers.Vectors A.18.2(21/5)
in Ada.Strings.Bounded A.4.4(9)
in Ada.Strings.Unbounded A.4.5(6)
in Ada.Text_IO.Editing F.3.3(11)
in Interfaces.COBOL B.4(34), B.4(39), B.4(44)
Less_Case_Insensitive
child of Ada.Strings A.4.10(13/5)
child of Ada.Strings.Bounded A.4.10(18/5)
child of Ada.Strings.Fixed A.4.10(16/3)
child of Ada.Strings.Unbounded A.4.10(21/5)
Line *in* Ada.Text_IO A.10.1(38/5)
Line_Length *in* Ada.Text_IO A.10.1(25/5)
Local_Image
in Ada.Calendar.Formatting 9.6.1(35.1/5)
Local_Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(6/5)
Log
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Look_Ahead *in* Ada.Text_IO A.10.1(43/5)
Max
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Maximum_Length
in Ada.Containers.Vectors A.18.2(12.3/5)
Meaningful_For
in Ada.Containers.Multiway_Trees A.18.10(22.1/5)
Members
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Merge
in Ada.Containers.Doubly_Linked_Lists A.18.3(50/5)
in Ada.Containers.Vectors A.18.2(78/5)
Microseconds *in* Ada.Real_Time D.8(14/2)
Milliseconds *in* Ada.Real_Time D.8(14/2)
Min
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Minute *in* Ada.Calendar.Formatting 9.6.1(25/2)
Minutes *in* Ada.Real_Time D.8(14/2)
Mode
in Ada.Direct_IO A.8.4(9)
in Ada.Sequential_IO A.8.1(9)
in Ada.Streams.Stream_IO A.12.1(11)
in Ada.Text_IO A.10.1(12)
Modification_Time *in* Ada.Directories A.16(27/2), A.16(42/2)
Modulus
in Ada.Numerics.Generic_Complex_Arrays G.3.2(10/2), G.3.2(30/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(9)
Month
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(22/2)
More_Entries *in* Ada.Directories A.16(34/2)
Move
in Ada.Containers.Doubly_Linked_Lists A.18.3(18/5)
in Ada.Containers.Hashed_Maps A.18.5(18/5)
in Ada.Containers.Hashed_Sets A.18.8(18/5)
in Ada.Containers.Indefinite_Holders A.18.18(22/5)
in Ada.Containers.Multiway_Trees A.18.10(34/5)
in Ada.Containers.Ordered_Maps A.18.6(17/5)
in Ada.Containers.Ordered_Sets A.18.9(17/5)
in Ada.Containers.Vectors A.18.2(35/5)
in Ada.Strings.Fixed A.4.3(7)
Name
in Ada.Direct_IO A.8.4(9), A.8.4(18.12/5), A.8.4(18.5/5)
in Ada.Sequential_IO A.8.1(9), A.8.1(15.12/5), A.8.1(15.5/5)
in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.12/5), A.12.1(26.5/5)
in Ada.Text_IO A.10.1(12), A.10.1(85.12/5), A.10.1(85.5/5)
Name_Case_Equivalence
in Ada.Directories A.16(20.2/3)
Nanoseconds *in* Ada.Real_Time D.8(14/2)
New_Char_Array
in Interfaces.C.Strings B.3.1(9)
New_Line
in Ada.Strings.Text_Buffers A.4.12(12/5)
in Ada.Text_IO A.10.1(28/5)
New_Page *in* Ada.Text_IO A.10.1(31/5)
New_String *in* Interfaces.C.Strings B.3.1(10)
New_Vector *in* Ada.Containers.Vectors A.18.2(14.1/5)
Next
in Ada.Containers.Doubly_Linked_Lists A.18.3(37.1/5), A.18.3(37/5), A.18.3(39.1/5), A.18.3(39/5)
in Ada.Containers.Hashed_Maps A.18.5(28.1/5), A.18.5(28/5), A.18.5(29.1/5), A.18.5(29/5)
in Ada.Containers.Hashed_Sets A.18.8(41.1/5), A.18.8(41/5), A.18.8(42.1/5), A.18.8(42/5)
in Ada.Containers.Ordered_Maps A.18.6(34.1/5), A.18.6(34/5), A.18.6(35.1/5), A.18.6(35/5)
in Ada.Containers.Ordered_Sets A.18.9(45.1/5), A.18.9(45/5), A.18.9(46.1/5), A.18.9(46/5)
in Ada.Containers.Vectors A.18.2(63.1/5), A.18.2(63/5), A.18.2(64.1/5), A.18.2(64/5)
in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.7/5)
Next_Sibling
in Ada.Containers.Multiway_Trees A.18.10(64.1/5), A.18.10(64/5), A.18.10(65.1/5), A.18.10(65/5)
Node_Count
in Ada.Containers.Multiway_Trees A.18.10(17/5)
Null_Task_Id
in Ada.Task_Identification C.7.1(2/5)

Number_Of_CPUs
in System.Multiprocessors D.16(5/3)
Numerator
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(7/5)
Open
in Ada.Direct_IO A.8.4(7), A.8.4(18.11/5), A.8.4(18.4/5)
in Ada.Sequential_IO A.8.1(7), A.8.1(15.11/5), A.8.1(15.4/5)
in Ada.Streams.Stream_IO A.12.1(9), A.12.1(26.11/5), A.12.1(26.4/5)
in Ada.Text_IO A.10.1(10), A.10.1(85.11/5), A.10.1(85.4/5)
Overlap
in Ada.Containers.Hashed_Sets A.18.8(38/2)
in Ada.Containers.Ordered_Sets A.18.9(39/2)
Overwrite
in Ada.Strings.Bounded A.4.4(62), A.4.4(63)
in Ada.Strings.Fixed A.4.3(27), A.4.3(28)
in Ada.Strings.Unbounded A.4.5(57), A.4.5(58)
Page *in* Ada.Text_IO A.10.1(39/5)
Page_Length *in* Ada.Text_IO A.10.1(26/5)
Parent
in Ada.Containers.Multiway_Trees A.18.10(59.1/5), A.18.10(59/5)
Parent_Tag *in* Ada.Tags 3.9(7.2/2)
Peak_Use
in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
Pic_String *in* Ada.Text_IO.Editing F.3.3(7)
Pool_of_Subpool
in System.Storage_Pools.Subpools 13.11.4(9/3)
Prepend
in Ada.Containers.Doubly_Linked_Lists A.18.3(22/5)
in Ada.Containers.Vectors A.18.2(45/5)
Prepend_Child
in Ada.Containers.Multiway_Trees A.18.10(51/5)
Prepend_Vector
in Ada.Containers.Vectors A.18.2(44/5)
Previous
in Ada.Containers.Doubly_Linked_Lists A.18.3(38.1/5), A.18.3(38/5), A.18.3(40.1/5), A.18.3(40/5)
in Ada.Containers.Ordered_Maps A.18.6(36.1/5), A.18.6(36/5), A.18.6(37.1/5), A.18.6(37/5)
in Ada.Containers.Ordered_Sets A.18.9(47.1/5), A.18.9(47/5), A.18.9(48.1/5), A.18.9(48/5)
in Ada.Containers.Vectors A.18.2(65.1/5), A.18.2(65/5), A.18.2(66.1/5), A.18.2(66/5)
in Ada.Iterator_Interfaces 5.5.1(4/3)
Previous_Sibling
in Ada.Containers.Multiway_Trees A.18.10(66.1/5), A.18.10(66/5), A.18.10(67.1/5), A.18.10(67/5)
Put
in Ada.Strings.Text_Buffers A.4.12(7/5)
in Ada.Text_IO A.10.1(42/5), A.10.1(48/5), A.10.1(55/5), A.10.1(60/5), A.10.1(66/5), A.10.1(67/5), A.10.1(71/5), A.10.1(72/5), A.10.1(76/5), A.10.1(77/5), A.10.1(82/5), A.10.1(83/5)
in Ada.Text_IO.Bounded_IO A.10.11(4/2), A.10.11(5/2)
in Ada.Text_IO.Complex_IO G.1.3(7), G.1.3(8/5)
in Ada.Text_IO.Editing F.3.3(14/5), F.3.3(15/5), F.3.3(16)
in Ada.Text_IO.Unbounded_IO A.10.12(4/2), A.10.12(5/2)
Put_Image
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(17/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(20/5)
Put_Line
in Ada.Text_IO A.10.1(50/5)
in Ada.Text_IO.Bounded_IO A.10.11(6/2), A.10.11(7/2)
in Ada.Text_IO.Unbounded_IO A.10.12(6/2), A.10.12(7/2)
Put_UTF_8 *in* Ada.Strings.Text_Buffers A.4.12(10/5)
Query_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(16.1/5), A.18.3(16/5)
in Ada.Containers.Hashed_Maps A.18.5(16.1/5), A.18.5(16/5)
in Ada.Containers.Hashed_Sets A.18.8(17.1/5), A.18.8(17/5)
in Ada.Containers.Indefinite_Holders A.18.18(14/5)
in Ada.Containers.Multiway_Trees A.18.10(26.1/5), A.18.10(26/5)
in Ada.Containers.Ordered_Maps A.18.6(15.1/5), A.18.6(15/5)
in Ada.Containers.Ordered_Sets A.18.9(16.1/5), A.18.9(16/5)
in Ada.Containers.Vectors A.18.2(31/5), A.18.2(32.1/5), A.18.2(32/5)
Raise_Exception *in* Ada.Exceptions 11.4.1(4/3)
Random
in Ada.Numerics.Discrete_Random A.5.2(20/5)
in Ada.Numerics.Float_Random A.5.2(8/5)
Re
in Ada.Numerics.Generic_Complex_Arrays G.3.2(7/2), G.3.2(27/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(6)
Read
in Ada.Direct_IO A.8.4(12/5)
in Ada.Sequential_IO A.8.1(12/5)
in Ada.Storage_IO A.9(6)
in Ada.Streams 13.13.1(5)
in Ada.Streams.Storage.Bounded 13.13.1(28/5)
in Ada.Streams.Storage.Unbounded 13.13.1(20/5)
in Ada.Streams.Stream_IO A.12.1(15/5), A.12.1(16/5)
in System.RPC E.5(7)
Reference
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.4/5)
in Ada.Containers.Hashed_Maps A.18.5(17.4/5), A.18.5(17.6/5)
in Ada.Containers.Indefinite_Holders A.18.18(19/5)
in Ada.Containers.Multiway_Trees A.18.10(31/5)
in Ada.Containers.Ordered_Maps A.18.6(16.4/5), A.18.6(16.6/5)
in Ada.Containers.Vectors A.18.2(34.4/5), A.18.2(34.6/5)
in Ada.Interrupts C.3.2(10)
in Ada.Task_Attributes C.7.2(5)

Reference_Preserving_Key
in Ada.Containers.Hashed_Sets A.18.8(58.2/5), A.18.8(58.4/5)
in Ada.Containers.Ordered_Sets A.18.9(73.2/5), A.18.9(73.4/5)
Reinitialize *in* Ada.Task_Attributes C.7.2(6)
Relative_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(13/3)
Remove_Task
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Rename *in* Ada.Directories A.16(12/2)
Replace
in Ada.Containers.Hashed_Maps A.18.5(23/5)
in Ada.Containers.Hashed_Sets A.18.8(22/5), A.18.8(53/5)
in Ada.Containers.Ordered_Maps A.18.6(22/5)
in Ada.Containers.Ordered_Sets A.18.9(21/5), A.18.9(66/5)
Replace_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(15/5)
in Ada.Containers.Hashed_Maps A.18.5(15/5)
in Ada.Containers.Hashed_Sets A.18.8(16/5)
in Ada.Containers.Indefinite_Holders A.18.18(13/5)
in Ada.Containers.Multiway_Trees A.18.10(25/5)
in Ada.Containers.Ordered_Maps A.18.6(14/5)
in Ada.Containers.Ordered_Sets A.18.9(15/5)
in Ada.Containers.Vectors A.18.2(29/5), A.18.2(30/5)
in Ada.Strings.Bounded A.4.4(27)
in Ada.Strings.Unbounded A.4.5(21)
Replace_Slice
in Ada.Containers.Vectors A.18.2(53.3/6), A.18.2(53.4/6)
in Ada.Strings.Bounded A.4.4(58), A.4.4(59)
in Ada.Strings.Fixed A.4.3(23), A.4.3(24)
in Ada.Strings.Unbounded A.4.5(53), A.4.5(54)
Replenish
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Replicate *in* Ada.Strings.Bounded A.4.4(78), A.4.4(79), A.4.4(80)
Reraise_Occurrence *in* Ada.Exceptions 11.4.1(4/3)
Reserve_Capacity
in Ada.Containers.Hashed_Maps A.18.5(9/5)
in Ada.Containers.Hashed_Sets A.18.8(11/5)
in Ada.Containers.Vectors A.18.2(20/5)
Reset
in Ada.Direct_IO A.8.4(8)
in Ada.Numerics.Discrete_Random A.5.2(21/5), A.5.2(24/5)
in Ada.Numerics.Float_Random A.5.2(9/5), A.5.2(12/5)
in Ada.Sequential_IO A.8.1(8)
in Ada.Streams.Stream_IO A.12.1(10)
in Ada.Text_IO A.10.1(11)
Reverse_Elements
in Ada.Containers.Doubly_Linked_Lists A.18.3(27/5)
in Ada.Containers.Vectors A.18.2(54/5)
Reverse_Find
in Ada.Containers.Doubly_Linked_Lists A.18.3(42/5)
in Ada.Containers.Vectors A.18.2(70/5)
Reverse_Find_Index
in Ada.Containers.Vectors A.18.2(69/2)
Reverse_Iterate
in Ada.Containers.Doubly_Linked_Lists A.18.3(46/5)
in Ada.Containers.Ordered_Maps A.18.6(51/5)
in Ada.Containers.Ordered_Sets A.18.9(61/5)
in Ada.Containers.Vectors A.18.2(74/5)
Reverse_Iterate_Children
in Ada.Containers.Multiway_Trees A.18.10(69.1/5), A.18.10(69/5)
Root *in* Ada.Containers.Multiway_Trees A.18.10(22/5)
Save
in Ada.Numerics.Discrete_Random A.5.2(24/5)
in Ada.Numerics.Float_Random A.5.2(12/5)
Save_Occurrence *in* Ada.Exceptions 11.4.1(6/2)
Second *in* Ada.Calendar.Formatting 9.6.1(26/2)
Seconds
in Ada.Calendar 9.6(13)
in Ada.Real_Time D.8(14/2)
Seconds_Of *in* Ada.Calendar.Formatting 9.6.1(23.1/6), 9.6.1(28/2)
Set *in* Ada.Environment_Variables A.17(6/2)
Set_Bounded_String
in Ada.Strings.Bounded A.4.4(12.1/2)
Set_Col *in* Ada.Text_IO A.10.1(35/5)
Set_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(12/3)
Set_Deadline *in* Ada.Dispatching.EDF D.2.6(9/5)
Set_Dependents_Fallback_Handler
in Ada.Task_Termination C.7.3(5/2)
Set_Directory *in* Ada.Directories A.16(6/2)
Set_Error *in* Ada.Text_IO A.10.1(15)
Set_Exit_Status *in* Ada.Command_Line A.15(9)
Set_False
in Ada.Synchronous_Task_Control D.10(4/5)
Set_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Real_Time.Timing_Events D.15(5/2)
Set_Im
in Ada.Numerics.Generic_Complex_Arrays G.3.2(8/2), G.3.2(28/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(7)
Set_Index
in Ada.Direct_IO A.8.4(14/5)
in Ada.Streams.Stream_IO A.12.1(22/5)
Set_Input *in* Ada.Text_IO A.10.1(15)
Set_Length *in* Ada.Containers.Vectors A.18.2(22/5)
Set_Line *in* Ada.Text_IO A.10.1(36/5)
Set_Line_Length *in* Ada.Text_IO A.10.1(23/5)
Set_Mode *in* Ada.Streams.Stream_IO A.12.1(24)
Set_Output *in* Ada.Text_IO A.10.1(15)
Set_Page_Length *in* Ada.Text_IO A.10.1(24/5)
Set_Pool_of_Subpool
in System.Storage_Pools.Subpools 13.11.4(10/5)
Set_Priority
in Ada.Dynamic_Priorities D.5.1(4)
Set_Quantum
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Set_Re
in Ada.Numerics.Generic_Complex_Arrays G.3.2(8/2), G.3.2(28/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(7)
Set_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)

Set_Specific_Handler
in Ada.Task_Termination C.7.3(6/2)
Set_True
in Ada.Synchronous_Task_Control D.10(4/5)
Set_Unbounded_String
in Ada.Strings.Unbounded A.4.5(11.1/2)
Set_Value *in* Ada.Task_Attributes C.7.2(6)
Simple_Name
in Ada.Directories A.16(16/5), A.16(38/2)
in Ada.Directories.Hierarchical_File_Names A.16.1(10/3)
Sin
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Sinh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Size
in Ada.Direct_IO A.8.4(15)
in Ada.Directories A.16(26/2), A.16(41/2)
in Ada.Streams.Stream_IO A.12.1(23)
Skip_Line *in* Ada.Text_IO A.10.1(29/5)
Skip_Page *in* Ada.Text_IO A.10.1(32/5)
Slice
in Ada.Containers.Vectors A.18.2(53.1/6), A.18.2(53.2/6)
in Ada.Strings.Bounded A.4.4(28)
in Ada.Strings.Unbounded A.4.5(22)
Solve
in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Sort
in Ada.Containers.Doubly_Linked_Lists A.18.3(49/5)
in Ada.Containers.Vectors A.18.2(77/5)
Specific_Handler
in Ada.Task_Termination C.7.3(6/2)
Splice
in Ada.Containers.Doubly_Linked_Lists A.18.3(30/5), A.18.3(31/5), A.18.3(32/2)
Splice_Children
in Ada.Containers.Multiway_Trees A.18.10(57/5), A.18.10(58/5)
Splice_Subtree
in Ada.Containers.Multiway_Trees A.18.10(55/5), A.18.10(56/5)
Split
in Ada.Calendar 9.6(14)
in Ada.Calendar.Formatting 9.6.1(29/2), 9.6.1(32/2), 9.6.1(33/2), 9.6.1(34/2)
in Ada.Execution_Time D.14(8/2)
in Ada.Real_Time D.8(16)
Split_Into_Chunks
in Ada.Iterator_Interfaces 5.5.1(4.4/5)
Sqrt
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Standard_Error *in* Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Input *in* Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Output *in* Ada.Text_IO A.10.1(16), A.10.1(19)
Start_Search *in* Ada.Directories A.16(32/2)
Storage_Size
in System.Storage_Pools 13.11(9)
in System.Storage_Pools.Subpools 13.11.4(16/3)
Stream
in Ada.Streams.Stream_IO A.12.1(13)
in Ada.Text_IO.Text_Streams A.12.2(4)
in Ada.Wide_Text_IO.Text_Streams A.12.3(4)
in Ada.Wide_Wide_Text_IO.Text_Streams A.12.4(4/2)
Strlen *in* Interfaces.C.Strings B.3.1(17)
Sub_Second *in* Ada.Calendar.Formatting 9.6.1(27/2)
Subtree_Node_Count
in Ada.Containers.Multiway_Trees A.18.10(18.1/5), A.18.10(18/5)
Supported
in Ada.Execution_Time.Interrupts D.14.3(3/5)
Suspend_Until_True
in Ada.Synchronous_Task_Control D.10(4/5)
Suspend_Until_True_And_Set_Deadline
in Ada.Synchronous_Task_Control.EDF D.10(5.2/5)
Swap
in Ada.Containers.Doubly_Linked_Lists A.18.3(28/5)
in Ada.Containers.Indefinite_Holders A.18.18(22.1/5)
in Ada.Containers.Multiway_Trees A.18.10(37/5)
in Ada.Containers.Vectors A.18.2(55/5), A.18.2(56/5)
Swap_Links
in Ada.Containers.Doubly_Linked_Lists A.18.3(29/5)
Symmetric_Difference
in Ada.Containers.Hashed_Sets A.18.8(35/5), A.18.8(36/5)
in Ada.Containers.Ordered_Sets A.18.9(36/5), A.18.9(37/5)
Tail
in Ada.Strings.Bounded A.4.4(72), A.4.4(73)
in Ada.Strings.Fixed A.4.3(37), A.4.3(38)
in Ada.Strings.Unbounded A.4.5(67), A.4.5(68)
Tampering_With_Cursors_Prohibited
in Ada.Containers.Doubly_Linked_Lists A.18.3(10.1/5)
in Ada.Containers.Hashed_Maps A.18.5(7.1/5)
in Ada.Containers.Hashed_Sets A.18.8(8.1/5)
in Ada.Containers.Multiway_Trees A.18.10(15.1/5)
in Ada.Containers.Ordered_Maps A.18.6(8.1/5)
in Ada.Containers.Ordered_Sets A.18.9(9.1/5)
in Ada.Containers.Vectors A.18.2(12.1/5)
Tampering_With_Elements_Prohibited
in Ada.Containers.Doubly_Linked_Lists A.18.3(10.2/5)
in Ada.Containers.Hashed_Maps A.18.5(7.2/5)
in Ada.Containers.Multiway_Trees A.18.10(15.2/5)
in Ada.Containers.Ordered_Maps A.18.6(8.2/5)
in Ada.Containers.Vectors A.18.2(12.2/5)
Tampering_With_The_Element_Prohibited
in Ada.Containers.Indefinite_Holders A.18.18(8.1/5)
Tan
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Tanh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Text_Truncated
in Ada.Strings.Text_Buffers.Bounded A.4.12(28/5)

Time_Of
in Ada.Calendar 9.6(15)
in Ada.Calendar.Formatting 9.6.1(30/2), 9.6.1(31/2)
in Ada.Execution_Time D.14(9/2)
in Ada.Real_Time D.8(16)
Time_Of_Event
in Ada.Real_Time.Timing_Events D.15(6/2)
Time_Remaining
in Ada.Execution_Time.Timers D.14.1(8/2)
To_Ada
in Interfaces.C B.3(22), B.3(26), B.3(28), B.3(32), B.3(37), B.3(39), B.3(39.10/2), B.3(39.13/2), B.3(39.17/2), B.3(39.19/2), B.3(39.4/2), B.3(39.8/2)
in Interfaces.COBOL B.4(17), B.4(19)
in Interfaces.Fortran B.5(13), B.5(14), B.5(16)
To_Address
in System.Address_To_Access_Conversions 13.7.2(3/3)
in System.Storage_Elements 13.7.1(10/3)
To_Basic
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20.1/5), A.3.5(21.1/5)
To_Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(7/5), A.5.6(12/5), A.5.6(13/5)
To_Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(9/5), A.5.7(13/5), A.5.7(14.1/6), A.5.7(14/5)
To_Binary *in* Interfaces.COBOL B.4(45), B.4(48)
To_Bounded_String
in Ada.Strings.Bounded A.4.4(11)
To_C *in* Interfaces.C B.3(21), B.3(25), B.3(27), B.3(32), B.3(36), B.3(38), B.3(39.13/2), B.3(39.16/2), B.3(39.18/2), B.3(39.4/2), B.3(39.7/2), B.3(39.9/2)
To_Character
in Ada.Characters.Conversions A.3.4(5/2)
To_Chars_Ptr *in* Interfaces.C.Strings B.3.1(8)
To_COBOL *in* Interfaces.COBOL B.4(17), B.4(18)
To_Cursor *in* Ada.Containers.Vectors A.18.2(25/5)
To_Decimal *in* Interfaces.COBOL B.4(35), B.4(40), B.4(44), B.4(47)
To_Display *in* Interfaces.COBOL B.4(36)
To_Domain
in Ada.Strings.Maps A.4.2(24)
in Ada.Strings.Wide_Maps A.4.7(24)
in Ada.Strings.Wide_Wide_Maps A.4.8(24/2)
To_Duration *in* Ada.Real_Time D.8(13)
To_Fortran *in* Interfaces.Fortran B.5(13), B.5(14), B.5(15)
To_Holder
in Ada.Containers.Indefinite_Holders A.18.18(9/5)
To_Index *in* Ada.Containers.Vectors A.18.2(26.1/5), A.18.2(26/5)
To_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(11/5)
in System.Storage_Elements 13.7.1(10/3)
To_ISO_646 *in* Ada.Characters.Handling A.3.2(11), A.3.2(12)
To_Long_Binary *in* Interfaces.COBOL B.4(48)
To_Lower
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Mapping
in Ada.Strings.Maps A.4.2(23)
in Ada.Strings.Wide_Maps A.4.7(23)
in Ada.Strings.Wide_Wide_Maps A.4.8(23/2)
To_Packed *in* Interfaces.COBOL B.4(41)
To_Picture *in* Ada.Text_IO.Editing F.3.3(6)
To_Pointer
in System.Address_To_Access_Conversions 13.7.2(3/3)
To_Quotient_String
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
To_Range
in Ada.Strings.Maps A.4.2(24)
in Ada.Strings.Wide_Maps A.4.7(25)
in Ada.Strings.Wide_Wide_Maps A.4.8(25/2)
To_Ranges
in Ada.Strings.Maps A.4.2(10)
in Ada.Strings.Wide_Maps A.4.7(10)
in Ada.Strings.Wide_Wide_Maps A.4.8(10/2)
To_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(10/5)
To_Sequence
in Ada.Strings.Maps A.4.2(19)
in Ada.Strings.Wide_Maps A.4.7(19)
in Ada.Strings.Wide_Wide_Maps A.4.8(19/2)
To_Set
in Ada.Containers.Hashed_Sets A.18.8(9/5)
in Ada.Containers.Ordered_Sets A.18.9(10/5)
in Ada.Strings.Maps A.4.2(8), A.4.2(9), A.4.2(17), A.4.2(18)
in Ada.Strings.Wide_Maps A.4.7(8), A.4.7(9), A.4.7(17), A.4.7(18)
in Ada.Strings.Wide_Wide_Maps A.4.8(8/2), A.4.8(9/2), A.4.8(17/2), A.4.8(18/2)
To_String
in Ada.Characters.Conversions A.3.4(5/2)
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(14/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(15/5)
in Ada.Strings.Bounded A.4.4(12)
in Ada.Strings.Unbounded A.4.5(11)
To_Time_Span *in* Ada.Real_Time D.8(13)
To_Unbounded_String
in Ada.Strings.Unbounded A.4.5(9), A.4.5(10)
To_Upper
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Vector *in* Ada.Containers.Vectors A.18.2(13/5), A.18.2(14/5)
To_Wide_Character
in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_String
in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_Wide_Character
in Ada.Characters.Conversions A.3.4(4/2)
To_Wide_Wide_String
in Ada.Characters.Conversions A.3.4(4/2)
Translate
in Ada.Strings.Bounded A.4.4(53), A.4.4(54), A.4.4(55/6), A.4.4(56/6)
in Ada.Strings.Fixed A.4.3(18), A.4.3(19), A.4.3(20/6), A.4.3(21/6)

in Ada.Strings.Unbounded A.4.5(48), A.4.5(49), A.4.5(50/6), A.4.5(51/6)
Transpose
in Ada.Numerics.Generic_Complex_Arrays G.3.2(34/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(17/2)
Trim
in Ada.Strings.Bounded A.4.4(67), A.4.4(68), A.4.4(69)
in Ada.Strings.Fixed A.4.3(31), A.4.3(32), A.4.3(33), A.4.3(34)
in Ada.Strings.Unbounded A.4.5(61), A.4.5(62), A.4.5(63), A.4.5(64)
Unbounded_Slice
in Ada.Strings.Unbounded A.4.5(22.1/2), A.4.5(22.2/2)
Unchecked_Conversion
child of Ada 13.9(3/5)
Unchecked_Deallocation
child of Ada 13.11.2(3/5)
Union
in Ada.Containers.Hashed_Sets A.18.8(26/5), A.18.8(27/5)
in Ada.Containers.Ordered_Sets A.18.9(27/5), A.18.9(28/5)
Unit_Matrix
in Ada.Numerics.Generic_Complex_Arrays G.3.2(51/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(29/2)
Unit_Vector
in Ada.Numerics.Generic_Complex_Arrays G.3.2(24/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(14/2)
Update *in* Interfaces.C.Strings B.3.1(18), B.3.1(19)
Update_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(17/5)
in Ada.Containers.Hashed_Maps A.18.5(17/5)
in Ada.Containers.Indefinite_Holders A.18.18(15/5)
in Ada.Containers.Multiway_Trees A.18.10(27/5)
in Ada.Containers.Ordered_Maps A.18.6(16/5)
in Ada.Containers.Vectors A.18.2(33/5), A.18.2(34/5)
Update_Element_Preserving_Key
in Ada.Containers.Hashed_Sets A.18.8(58/5)
in Ada.Containers.Ordered_Sets A.18.9(73/5)
Update_Error *in* Interfaces.C.Strings B.3.1(20)
UTC_Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(6.1/5)
Valid
in Ada.Text_IO.Editing F.3.3(5), F.3.3(12)
in Interfaces.COBOL B.4(33), B.4(38), B.4(43)
Value
in Ada.Calendar.Formatting 9.6.1(36/2), 9.6.1(38/2)
in Ada.Environment_Variables A.17(4.1/3), A.17(4/2)
in Ada.Numerics.Discrete_Random A.5.2(26)
in Ada.Numerics.Float_Random A.5.2(14)
in Ada.Strings.Maps A.4.2(21)
in Ada.Strings.Wide_Maps A.4.7(21)
in Ada.Strings.Wide_Wide_Maps A.4.8(21/2)
in Ada.Task_Attributes C.7.2(4)
in Interfaces.C.Pointers B.3.2(6), B.3.2(7)
in Interfaces.C.Strings B.3.1(13), B.3.1(14), B.3.1(15), B.3.1(16)
Virtual_Length
in Interfaces.C.Pointers B.3.2(13)
Wait_For_Release
in Ada.Synchronous_Barriers D.10.1(6/5)
Wide_Equal_Case_Insensitive
child of Ada.Strings A.4.7(1/3)
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Hash
child of Ada.Strings A.4.7(1/3)
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Hash_Case_Insensitive
child of Ada.Strings A.4.7(1/3)
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Exception_Name *in* Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Expanded_Name *in* Ada.Tags 3.9(7/2)
Wide_Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(21/5)
Wide_Get_UTF_16
in Ada.Strings.Text_Buffers.Unbounded A.4.12(24/5)
Wide_Put *in* Ada.Strings.Text_Buffers A.4.12(8/5)
Wide_Put_UTF_16
in Ada.Strings.Text_Buffers A.4.12(11/5)
Wide_Wide_Equal_Case_Insensitive
child of Ada.Strings A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Wide_Wide_Hash
child of Ada.Strings A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Wide_Wide_Hash_Case_Insensitive
child of Ada.Strings A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Wide_Wide_Exception_Name
in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Wide_Expanded_Name *in* Ada.Tags 3.9(7/2)
Wide_Wide_Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(22/5)
Wide_Wide_Put
in Ada.Strings.Text_Buffers A.4.12(9/5)
Write
in Ada.Direct_IO A.8.4(13/5)
in Ada.Sequential_IO A.8.1(12/5)
in Ada.Storage_IO A.9(7)
in Ada.Streams 13.13.1(6)
in Ada.Streams.Storage.Bounded 13.13.1(29/5)
in Ada.Streams.Storage.Unbounded 13.13.1(21/5)
in Ada.Streams.Stream_IO A.12.1(18/5), A.12.1(19/5)
in System.RPC E.5(8)
Year
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(21/2)
Yield *in* Ada.Dispatching D.2.1(1.3/5)

Yield_To_Higher
  in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)

Yield_To_Same_Or_Higher
  in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)

Q.4 Language-Defined Exceptions

This subclause lists all language-defined exceptions. 1/3

Argument_Error
  in Ada.Numerics A.5(3/5)
Assertion_Error
  in Ada.Assertions 11.4.2(13/2)
Capacity_Error
  in Ada.Containers A.18.1(5.1/3)
Communication_Error
  in System.RPC E.5(5)
Constraint_Error
  in Standard A.1(46)
Conversion_Error
  in Interfaces.COBOL B.4(30)
Data_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Storage_IO A.9(9)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Device_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Dispatching_Domain_Error
  in System.Multiprocessors.Dispatching_Domains D.16.1(4/3)
Dispatching_Policy_Error
  in Ada.Dispatching D.2.1(1.4/3)
Encoding_Error
  in Ada.Strings.UTF_Encoding A.4.11(8/3)
End_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Group_Budget_Error
  in Ada.Execution_Time.Group_Budgets D.14.2(11/2)
Index_Error
  in Ada.Strings A.4.1(5)
Layout_Error
  in Ada.IO_Exceptions A.13(4)
  in Ada.Text_IO A.10.1(85/5)
Length_Error
  in Ada.Strings A.4.1(5)
Mode_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Name_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Pattern_Error
  in Ada.Strings A.4.1(5)
Picture_Error
  in Ada.Text_IO.Editing F.3.3(9)
Pointer_Error
  in Interfaces.C.Pointers B.3.2(8)
Program_Error
  in Standard A.1(46)
Status_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Storage_Error
  in Standard A.1(46)
Tag_Error
  in Ada.Tags 3.9(8)
Tasking_Error
  in Standard A.1(46)
Terminator_Error
  in Interfaces.C B.3(40)
Time_Error
  in Ada.Calendar 9.6(18)
Timer_Resource_Error
  in Ada.Execution_Time.Timers D.14.1(9/2)
Translation_Error
  in Ada.Strings A.4.1(5)
Unknown_Zone_Error
  in Ada.Calendar.Time_Zones 9.6.1(5/2)
Use_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)

in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)

Q.5 Language-Defined Objects

1/3 This subclause lists all language-defined constants, variables, named numbers, and enumeration literals.

ACK *in* Ada.Characters.Latin_1 A.3.3(5)
Acute *in* Ada.Characters.Latin_1 A.3.3(22)
Ada_To_COBOL *in* Interfaces.COBOL B.4(14)
Alphanumeric_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Ampersand *in* Ada.Characters.Latin_1 A.3.3(8)
APC *in* Ada.Characters.Latin_1 A.3.3(19)
Apostrophe *in* Ada.Characters.Latin_1 A.3.3(8)
Asterisk *in* Ada.Characters.Latin_1 A.3.3(8)
Basic_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Basic_Set
in Ada.Strings.Maps.Constants A.4.6(4)
BEL *in* Ada.Characters.Latin_1 A.3.3(5)
BOM_16 *in* Ada.Strings.UTF_Encoding A.4.11(12/3)
BOM_16BE *in* Ada.Strings.UTF_Encoding A.4.11(10/3)
BOM_16LE *in* Ada.Strings.UTF_Encoding A.4.11(11/3)
BOM_8 *in* Ada.Strings.UTF_Encoding A.4.11(9/3)
BPH *in* Ada.Characters.Latin_1 A.3.3(17)
Broken_Bar *in* Ada.Characters.Latin_1 A.3.3(21/3)
BS *in* Ada.Characters.Latin_1 A.3.3(5)
Buffer_Size *in* Ada.Storage_IO A.9(4)
CAN *in* Ada.Characters.Latin_1 A.3.3(6)
CCH *in* Ada.Characters.Latin_1 A.3.3(18)
Cedilla *in* Ada.Characters.Latin_1 A.3.3(22)
Cent_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)
char16_nul *in* Interfaces.C B.3(39.3/2)
char32_nul *in* Interfaces.C B.3(39.12/2)
CHAR_BIT *in* Interfaces.C B.3(6)
Character_Set
in Ada.Strings.Wide_Maps A.4.7(46/2)
in Ada.Strings.Wide_Maps.Wide_Constants A.4.8(48/2)
Circumflex *in* Ada.Characters.Latin_1 A.3.3(12)
COBOL_To_Ada *in* Interfaces.COBOL B.4(15)
Colon *in* Ada.Characters.Latin_1 A.3.3(10)
Comma *in* Ada.Characters.Latin_1 A.3.3(8)
Commercial_At
in Ada.Characters.Latin_1 A.3.3(10)
Control_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Copyright_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Country_Unknown *in* Ada.Locales A.19(5/3)
CPU_Tick *in* Ada.Execution_Time D.14(4/2)
CPU_Time_First *in* Ada.Execution_Time D.14(4/2)
CPU_Time_Last *in* Ada.Execution_Time D.14(4/2)
CPU_Time_Unit *in* Ada.Execution_Time D.14(4/2)
CR *in* Ada.Characters.Latin_1 A.3.3(5)
CSI *in* Ada.Characters.Latin_1 A.3.3(19)
Currency_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
DC1 *in* Ada.Characters.Latin_1 A.3.3(6)
DC2 *in* Ada.Characters.Latin_1 A.3.3(6)
DC3 *in* Ada.Characters.Latin_1 A.3.3(6)
DC4 *in* Ada.Characters.Latin_1 A.3.3(6)
DCS *in* Ada.Characters.Latin_1 A.3.3(18)
Decimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Default_Aft
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Base *in* Ada.Text_IO A.10.1(53), A.10.1(58)
Default_Bit_Order *in* System 13.7(15/2)
Default_Currency
in Ada.Text_IO.Editing F.3.3(10)
Default_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Default_Exp
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Fill *in* Ada.Text_IO.Editing F.3.3(10)
Default_Fore
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Priority *in* System 13.7(17)
Default_Quantum
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Default_Radix_Mark
in Ada.Text_IO.Editing F.3.3(10)
Default_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Default_Separator
in Ada.Text_IO.Editing F.3.3(10)
Default_Setting *in* Ada.Text_IO A.10.1(80)
Default_Width *in* Ada.Text_IO A.10.1(53), A.10.1(58), A.10.1(80)
Degree_Sign *in* Ada.Characters.Latin_1 A.3.3(22)
DEL *in* Ada.Characters.Latin_1 A.3.3(14)
Diaeresis *in* Ada.Characters.Latin_1 A.3.3(21/3)
Division_Sign
in Ada.Characters.Latin_1 A.3.3(26)
DLE *in* Ada.Characters.Latin_1 A.3.3(6)
Dollar_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
e *in* Ada.Numerics A.5(3/5)
EM *in* Ada.Characters.Latin_1 A.3.3(6)
Empty_Holder
in Ada.Containers.Indefinite_Holders A.18.18(7/3)

Empty_List
in Ada.Containers.Doubly_Linked_Lists A.18.3(8/2), A.18.3(51.4/5)
Empty_Map
in Ada.Containers.Hashed_Maps A.18.5(5/2), A.18.5(37.5/5)
in Ada.Containers.Ordered_Maps A.18.6(6/2), A.18.6(51.6/5)
Empty_Set
in Ada.Containers.Hashed_Sets A.18.8(5/2), A.18.8(59.4/5)
in Ada.Containers.Ordered_Sets A.18.9(6/2), A.18.9(74.4/5)
Empty_Tree
in Ada.Containers.Multiway_Trees A.18.10(10/3), A.18.10(70.4/5)
Empty_Vector
in Ada.Containers.Vectors A.18.2(10/2), A.18.2(79.4/5)
ENQ *in* Ada.Characters.Latin_1 A.3.3(5)
EOT *in* Ada.Characters.Latin_1 A.3.3(5)
EPA *in* Ada.Characters.Latin_1 A.3.3(18)
Equals_Sign *in* Ada.Characters.Latin_1 A.3.3(10)
ESA *in* Ada.Characters.Latin_1 A.3.3(17)
ESC *in* Ada.Characters.Latin_1 A.3.3(6)
ETB *in* Ada.Characters.Latin_1 A.3.3(6)
ETX *in* Ada.Characters.Latin_1 A.3.3(5)
Exclamation *in* Ada.Characters.Latin_1 A.3.3(8)
Failure *in* Ada.Command_Line A.15(8)
Feminine_Ordinal_Indicator
in Ada.Characters.Latin_1 A.3.3(21/3)
FF *in* Ada.Characters.Latin_1 A.3.3(5)
Fine_Delta *in* System 13.7(9)
Fraction_One_Half
in Ada.Characters.Latin_1 A.3.3(22)
Fraction_One_Quarter
in Ada.Characters.Latin_1 A.3.3(22)
Fraction_Three_Quarters
in Ada.Characters.Latin_1 A.3.3(22)
Friday *in* Ada.Calendar.Formatting 9.6.1(17/2)
FS *in* Ada.Characters.Latin_1 A.3.3(6)
Full_Stop *in* Ada.Characters.Latin_1 A.3.3(8)
Graphic_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Grave *in* Ada.Characters.Latin_1 A.3.3(13)
Greater_Than_Sign
in Ada.Characters.Latin_1 A.3.3(10)
GS *in* Ada.Characters.Latin_1 A.3.3(6)
Hexadecimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)
High_Order_First
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
HT *in* Ada.Characters.Latin_1 A.3.3(5)
HTJ *in* Ada.Characters.Latin_1 A.3.3(17)
HTS *in* Ada.Characters.Latin_1 A.3.3(17)
Hyphen *in* Ada.Characters.Latin_1 A.3.3(8)
i
in Ada.Numerics.Generic_Complex_Types G.1.1(5)
in Interfaces.Fortran B.5(10)
Identity
in Ada.Strings.Maps A.4.2(22)
in Ada.Strings.Wide_Maps A.4.7(22)
in Ada.Strings.Wide_Wide_Maps A.4.8(22/2)
Interrupt_Clocks_Supported
in Ada.Execution_Time D.14(9.1/3)
Inverted_Exclamation
in Ada.Characters.Latin_1 A.3.3(21/3)
Inverted_Question
in Ada.Characters.Latin_1 A.3.3(22)
IS1 *in* Ada.Characters.Latin_1 A.3.3(16)
IS2 *in* Ada.Characters.Latin_1 A.3.3(16)
IS3 *in* Ada.Characters.Latin_1 A.3.3(16)
IS4 *in* Ada.Characters.Latin_1 A.3.3(16)
ISO_646_Set
in Ada.Strings.Maps.Constants A.4.6(4)
j
in Ada.Numerics.Generic_Complex_Types G.1.1(5)
in Interfaces.Fortran B.5(10)
Language_Unknown *in* Ada.Locales A.19(5/3)
LC_A *in* Ada.Characters.Latin_1 A.3.3(13)
LC_A_Acute *in* Ada.Characters.Latin_1 A.3.3(25)
LC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Grave *in* Ada.Characters.Latin_1 A.3.3(25)
LC_A_Ring *in* Ada.Characters.Latin_1 A.3.3(25)
LC_A_Tilde *in* Ada.Characters.Latin_1 A.3.3(25)
LC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(25)
LC_B *in* Ada.Characters.Latin_1 A.3.3(13)
LC_C *in* Ada.Characters.Latin_1 A.3.3(13)
LC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(25)
LC_D *in* Ada.Characters.Latin_1 A.3.3(13)
LC_E *in* Ada.Characters.Latin_1 A.3.3(13)
LC_E_Acute *in* Ada.Characters.Latin_1 A.3.3(25)
LC_E_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Grave *in* Ada.Characters.Latin_1 A.3.3(25)
LC_F *in* Ada.Characters.Latin_1 A.3.3(13)
LC_G *in* Ada.Characters.Latin_1 A.3.3(13)
LC_German_Sharp_S
in Ada.Characters.Latin_1 A.3.3(24)
LC_H *in* Ada.Characters.Latin_1 A.3.3(13)
LC_I *in* Ada.Characters.Latin_1 A.3.3(13)
LC_I_Acute *in* Ada.Characters.Latin_1 A.3.3(25)
LC_I_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Grave *in* Ada.Characters.Latin_1 A.3.3(25)
LC_Icelandic_Eth
in Ada.Characters.Latin_1 A.3.3(26)
LC_Icelandic_Thorn
in Ada.Characters.Latin_1 A.3.3(26)
LC_J *in* Ada.Characters.Latin_1 A.3.3(13)
LC_K *in* Ada.Characters.Latin_1 A.3.3(13)
LC_L *in* Ada.Characters.Latin_1 A.3.3(13)
LC_M *in* Ada.Characters.Latin_1 A.3.3(13)

LC_N *in* Ada.Characters.Latin_1 A.3.3(13)
LC_N_Tilde *in* Ada.Characters.Latin_1 A.3.3(26)
LC_O *in* Ada.Characters.Latin_1 A.3.3(13)
LC_O_Acute *in* Ada.Characters.Latin_1 A.3.3(26)
LC_O_Circumflex
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Grave *in* Ada.Characters.Latin_1 A.3.3(26)
LC_O_Oblique_Stroke
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Tilde *in* Ada.Characters.Latin_1 A.3.3(26)
LC_P *in* Ada.Characters.Latin_1 A.3.3(14)
LC_Q *in* Ada.Characters.Latin_1 A.3.3(14)
LC_R *in* Ada.Characters.Latin_1 A.3.3(14)
LC_S *in* Ada.Characters.Latin_1 A.3.3(14)
LC_T *in* Ada.Characters.Latin_1 A.3.3(14)
LC_U *in* Ada.Characters.Latin_1 A.3.3(14)
LC_U_Acute *in* Ada.Characters.Latin_1 A.3.3(26)
LC_U_Circumflex
in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Grave *in* Ada.Characters.Latin_1 A.3.3(26)
LC_V *in* Ada.Characters.Latin_1 A.3.3(14)
LC_W *in* Ada.Characters.Latin_1 A.3.3(14)
LC_X *in* Ada.Characters.Latin_1 A.3.3(14)
LC_Y *in* Ada.Characters.Latin_1 A.3.3(14)
LC_Y_Acute *in* Ada.Characters.Latin_1 A.3.3(26)
LC_Y_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_Z *in* Ada.Characters.Latin_1 A.3.3(14)
Leading_Nonseparate
in Interfaces.COBOL B.4(23)
Leading_Separate *in* Interfaces.COBOL B.4(23)
Left_Angle_Quotation
in Ada.Characters.Latin_1 A.3.3(21/3)
Left_Curly_Bracket
in Ada.Characters.Latin_1 A.3.3(14)
Left_Parenthesis
in Ada.Characters.Latin_1 A.3.3(8)
Left_Square_Bracket
in Ada.Characters.Latin_1 A.3.3(12)
Less_Than_Sign
in Ada.Characters.Latin_1 A.3.3(10)
Letter_Set
in Ada.Strings.Maps.Constants A.4.6(4)
LF *in* Ada.Characters.Latin_1 A.3.3(5)
Low_Line *in* Ada.Characters.Latin_1 A.3.3(12)
Low_Order_First
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
Lower_Case_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Lower_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Macron *in* Ada.Characters.Latin_1 A.3.3(21/3)
Masculine_Ordinal_Indicator
in Ada.Characters.Latin_1 A.3.3(22)
Max_Base_Digits *in* System 13.7(8)
Max_Binary_Modulus *in* System 13.7(7)
Max_Decimal_Digits *in* Ada.Decimal F.2(5)
Max_Delta *in* Ada.Decimal F.2(4)
Max_Digits *in* System 13.7(8)
Max_Digits_Binary *in* Interfaces.COBOL B.4(11)
Max_Digits_Long_Binary
in Interfaces.COBOL B.4(11)
Max_Image_Width
in Ada.Numerics.Discrete_Random A.5.2(25)
in Ada.Numerics.Float_Random A.5.2(13)
Max_Int *in* System 13.7(6)
Max_Length *in* Ada.Strings.Bounded A.4.4(5)
Max_Mantissa *in* System 13.7(9)
Max_Nonbinary_Modulus *in* System 13.7(7)
Max_Picture_Length
in Ada.Text_IO.Editing F.3.3(8)
Max_Scale *in* Ada.Decimal F.2(3)
Memory_Size *in* System 13.7(13)
Micro_Sign *in* Ada.Characters.Latin_1 A.3.3(22)
Middle_Dot *in* Ada.Characters.Latin_1 A.3.3(22)
Min_Delta *in* Ada.Decimal F.2(4)
Min_Handler_Ceiling
in Ada.Execution_Time.Group_Budgets D.14.2(7/2)
in Ada.Execution_Time.Timers D.14.1(6/2)
Min_Int *in* System 13.7(6)
Min_Scale *in* Ada.Decimal F.2(3)
Minus_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
Monday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Multiplication_Sign
in Ada.Characters.Latin_1 A.3.3(24)
MW *in* Ada.Characters.Latin_1 A.3.3(18)
NAK *in* Ada.Characters.Latin_1 A.3.3(6)
Native_Binary *in* Interfaces.COBOL B.4(25)
NBH *in* Ada.Characters.Latin_1 A.3.3(17)
NBSP *in* Ada.Characters.Latin_1 A.3.3(21/3)
NEL *in* Ada.Characters.Latin_1 A.3.3(17)
New_Line_Count
in Ada.Strings.Text_Buffers A.4.12(5/5)
No_Break_Space
in Ada.Characters.Latin_1 A.3.3(21/3)
No_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(9/2), A.18.3(51.5/5)
in Ada.Containers.Hashed_Maps A.18.5(6/2), A.18.5(37.6/5)
in Ada.Containers.Hashed_Sets A.18.8(6/2), A.18.8(59.5/5)
in Ada.Containers.Multiway_Trees A.18.10(11/3), A.18.10(70.5/5)
in Ada.Containers.Ordered_Maps A.18.6(7/2), A.18.6(51.7/5)
in Ada.Containers.Ordered_Sets A.18.9(7/2), A.18.9(74.5/5)
in Ada.Containers.Vectors A.18.2(11/2), A.18.2(79.5/5)
No_Index *in* Ada.Containers.Vectors A.18.2(7/2)
No_Tag *in* Ada.Tags 3.9(6.1/2)
Not_A_Specific_CPU
in System.Multiprocessors D.16(4/3)
Not_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)
NUL
in Ada.Characters.Latin_1 A.3.3(5)
in Interfaces.C B.3(20/1)

Null_Address *in* System 13.7(12)
Null_Bounded_String
in Ada.Strings.Bounded A.4.4(7)
Null_Id *in* Ada.Exceptions 11.4.1(2/5)
Null_Occurrence *in* Ada.Exceptions 11.4.1(3/5)
Null_Ptr *in* Interfaces.C.Strings B.3.1(7)
Null_Set
in Ada.Strings.Maps A.4.2(5)
in Ada.Strings.Wide_Maps A.4.7(5)
in Ada.Strings.Wide_Wide_Maps A.4.8(5/2)
Null_Unbounded_String
in Ada.Strings.Unbounded A.4.5(5)
Number_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
OSC *in* Ada.Characters.Latin_1 A.3.3(19)
Packed_Signed *in* Interfaces.COBOL B.4(27)
Packed_Unsigned *in* Interfaces.COBOL B.4(27)
Paragraph_Sign
in Ada.Characters.Latin_1 A.3.3(22)
Percent_Sign
in Ada.Characters.Latin_1 A.3.3(8)
Pi *in* Ada.Numerics A.5(3/5)
Pilcrow_Sign
in Ada.Characters.Latin_1 A.3.3(22)
PLD *in* Ada.Characters.Latin_1 A.3.3(17)
PLU *in* Ada.Characters.Latin_1 A.3.3(17)
Plus_Minus_Sign
in Ada.Characters.Latin_1 A.3.3(22)
Plus_Sign *in* Ada.Characters.Latin_1 A.3.3(8)
PM *in* Ada.Characters.Latin_1 A.3.3(19)
Pound_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)
PU1 *in* Ada.Characters.Latin_1 A.3.3(18)
PU2 *in* Ada.Characters.Latin_1 A.3.3(18)
Question *in* Ada.Characters.Latin_1 A.3.3(10)
Quotation *in* Ada.Characters.Latin_1 A.3.3(8)
Registered_Trade_Mark_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Reserved_128
in Ada.Characters.Latin_1 A.3.3(17)
Reserved_129
in Ada.Characters.Latin_1 A.3.3(17)
Reserved_132
in Ada.Characters.Latin_1 A.3.3(17)
Reserved_153
in Ada.Characters.Latin_1 A.3.3(19)
Reverse_Solidus
in Ada.Characters.Latin_1 A.3.3(12)
RI *in* Ada.Characters.Latin_1 A.3.3(17)
Right_Angle_Quotation
in Ada.Characters.Latin_1 A.3.3(22)
Right_Curly_Bracket
in Ada.Characters.Latin_1 A.3.3(14)
Right_Parenthesis
in Ada.Characters.Latin_1 A.3.3(8)
Right_Square_Bracket
in Ada.Characters.Latin_1 A.3.3(12)
Ring_Above *in* Ada.Characters.Latin_1 A.3.3(22)
RS *in* Ada.Characters.Latin_1 A.3.3(6)
Saturday *in* Ada.Calendar.Formatting 9.6.1(17/2)
SCHAR_MAX *in* Interfaces.C B.3(6)
SCHAR_MIN *in* Interfaces.C B.3(6)
SCI *in* Ada.Characters.Latin_1 A.3.3(19)
Section_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Semicolon *in* Ada.Characters.Latin_1 A.3.3(10)
Separate_Interrupt_Clocks_Supported
in Ada.Execution_Time D.14(9.2/3)
SI *in* Ada.Characters.Latin_1 A.3.3(5)
SO *in* Ada.Characters.Latin_1 A.3.3(5)
Soft_Hyphen *in* Ada.Characters.Latin_1 A.3.3(21/3)
SOH *in* Ada.Characters.Latin_1 A.3.3(5)
Solidus *in* Ada.Characters.Latin_1 A.3.3(8)
SOS *in* Ada.Characters.Latin_1 A.3.3(19)
SPA *in* Ada.Characters.Latin_1 A.3.3(18)
Space
in Ada.Characters.Latin_1 A.3.3(8)
in Ada.Strings A.4.1(4/2)
Special_Set
in Ada.Strings.Maps.Constants A.4.6(4)
SS2 *in* Ada.Characters.Latin_1 A.3.3(17)
SS3 *in* Ada.Characters.Latin_1 A.3.3(17)
SSA *in* Ada.Characters.Latin_1 A.3.3(17)
ST *in* Ada.Characters.Latin_1 A.3.3(19)
Standard_Indent
in Ada.Strings.Text_Buffers A.4.12(13/5)
Storage_Unit *in* System 13.7(13)
STS *in* Ada.Characters.Latin_1 A.3.3(18)
STX *in* Ada.Characters.Latin_1 A.3.3(5)
SUB *in* Ada.Characters.Latin_1 A.3.3(6)
Success *in* Ada.Command_Line A.15(8)
Sunday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Superscript_One
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Three
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Two
in Ada.Characters.Latin_1 A.3.3(22)
SYN *in* Ada.Characters.Latin_1 A.3.3(6)
System_Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(6/3)
System_Name *in* System 13.7(4)
Thursday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Tick
in Ada.Real_Time D.8(6)
in System 13.7(10)
Tilde *in* Ada.Characters.Latin_1 A.3.3(14)
Time_First *in* Ada.Real_Time D.8(4)
Time_Last *in* Ada.Real_Time D.8(4)
Time_Span_First *in* Ada.Real_Time D.8(5)
Time_Span_Last *in* Ada.Real_Time D.8(5)
Time_Span_Unit *in* Ada.Real_Time D.8(5)
Time_Span_Zero *in* Ada.Real_Time D.8(5)
Time_Unit *in* Ada.Real_Time D.8(4)
Trailing_Nonseparate
in Interfaces.COBOL B.4(23)
Trailing_Separate *in* Interfaces.COBOL B.4(23)
Tuesday *in* Ada.Calendar.Formatting 9.6.1(17/2)
UC_A_Acute *in* Ada.Characters.Latin_1 A.3.3(23)
UC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)

UC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Grave *in* Ada.Characters.Latin_1 A.3.3(23)
UC_A_Ring *in* Ada.Characters.Latin_1 A.3.3(23)
UC_A_Tilde *in* Ada.Characters.Latin_1 A.3.3(23)
UC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(23)
UC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Acute *in* Ada.Characters.Latin_1 A.3.3(23)
UC_E_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Grave *in* Ada.Characters.Latin_1 A.3.3(23)
UC_I_Acute *in* Ada.Characters.Latin_1 A.3.3(23)
UC_I_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Grave *in* Ada.Characters.Latin_1 A.3.3(23)
UC_Icelandic_Eth
in Ada.Characters.Latin_1 A.3.3(24)
UC_Icelandic_Thorn
in Ada.Characters.Latin_1 A.3.3(24)
UC_N_Tilde *in* Ada.Characters.Latin_1 A.3.3(24)
UC_O_Acute *in* Ada.Characters.Latin_1 A.3.3(24)
UC_O_Circumflex
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Diaeresis
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Grave *in* Ada.Characters.Latin_1 A.3.3(24)
UC_O_Oblique_Stroke
in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Tilde *in* Ada.Characters.Latin_1 A.3.3(24)
UC_U_Acute *in* Ada.Characters.Latin_1 A.3.3(24)
UC_U_Circumflex
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Diaeresis
in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Grave *in* Ada.Characters.Latin_1 A.3.3(24)
UC_Y_Acute *in* Ada.Characters.Latin_1 A.3.3(24)
UCHAR_MAX *in* Interfaces.C B.3(6)
Unbounded *in* Ada.Text_IO A.10.1(5)
Unsigned *in* Interfaces.COBOL B.4(23)
Upper_Case_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Upper_Set
in Ada.Strings.Maps.Constants A.4.6(4)
US *in* Ada.Characters.Latin_1 A.3.3(6)
Vertical_Line
in Ada.Characters.Latin_1 A.3.3(14)
VT *in* Ada.Characters.Latin_1 A.3.3(5)
VTS *in* Ada.Characters.Latin_1 A.3.3(17)
Wednesday *in* Ada.Calendar.Formatting 9.6.1(17/2)
Wide_Character_Set
in Ada.Strings.Wide_Maps.Wide_Constants A.4.8(48/2)
wide_nul *in* Interfaces.C B.3(31/1)
Wide_Space *in* Ada.Strings A.4.1(4/2)
Wide_Wide_Space *in* Ada.Strings A.4.1(4/2)
Word_Size *in* System 13.7(13)
Yen_Sign *in* Ada.Characters.Latin_1 A.3.3(21/3)

Index

Index entries are given by paragraph number.

& operator 4.5(1), 4.5.3(3)

* operator 4.5(1), 4.5.5(1)
** operator 4.5(1), 4.5.6(7)

+ operator 4.5(1), 4.5.3(1), 4.5.4(1)

- operator 4.5(1), 4.5.3(1), 4.5.4(1)

/ operator 4.5(1), 4.5.5(1)
/= operator 4.5(1), 4.5.2(1), 6.6(6/3)

10646:2020, ISO/IEC standard 1.2(8/6), 1.2.1(8.1/6)
14882:2020, ISO/IEC standard 1.2.1(9/5)
1539-1:2018, ISO/IEC standard 1.2.1(3/5)
1989:2002, ISO standard 1.2.1(4/5)

3166-1:2020, ISO standard 1.2(4.1/5)

60559:2020, ISO/IEC standard 1.2.1(10/5)
639-3:2007, ISO standard 1.2(1.1/3)
6429:1992, ISO/IEC standard 1.2.1(5/5)
646:1991, ISO/IEC standard 1.2.1(2/5)

8601-1:2019, ISO standard 1.2.1(6/5)
8859-1:1998, ISO/IEC standard 1.2.1(7/5)

9899:2018, ISO/IEC standard 1.2.1(8/5)

< operator 4.5(1), 4.5.2(1)
<= operator 4.5(1), 4.5.2(1)

= operator 4.5(1), 4.5.2(1)

> operator 4.5(1), 4.5.2(1)
>= operator 4.5(1), 4.5.2(1)

A

AARM 0.2(5/3)
abnormal completion 7.6.1(2/2)
abnormal state of an object 13.9.1(4)
  [*partial*] 9.8(21), 11.6(6/5), A.13(17)
abnormal task 9.8(4)
abort
  of a partition E.1(7)
  of a task 9.8(4)
  of the execution of a construct 9.8(5)
abort completion point 9.8(15)
abort-deferred operation 9.8(5)
abort_statement 9.8(2)
  used 5.1(4/2), P.1
Abort_Task
  in Ada.Task_Identification C.7.1(3/5)
abortable_part 9.7.4(5)
  used 9.7.4(2), P.1
abs operator 4.5(1), 4.5.6(1)
absolute deadline
  active D.2.6(14/5)
  base D.2.6(14/5)
absolute value 4.5(1), 4.5.6(1)
abstract data type (ADT)
  See private types and private extensions 7.3(1)
  See also abstract type 3.9.3(1/2)
abstract subprogram 3.9.3(1/2), 3.9.3(3/2)
abstract type 1.3.1(1/5), 3.9.3(1.2/2), 3.9.3(1/2)
abstract_subprogram_declaration 3.9.3(1.1/3)
  used 3.1(3/3), P.1
accept_alternative 9.7.1(5)
  used 9.7.1(4), P.1
accept_statement 9.5.2(3)
  used 5.1(5/5), 9.7.1(5), P.1
acceptable interpretation 8.6(14)
Access attribute 3.10.2(24/1), 3.10.2(32/5)
  See also Unchecked_Access attribute 13.10(3)
access discriminant 3.7(9/2)
access parameter 6.1(24/2)
access paths
  distinct 6.2(12/3)
access result type 6.1(24/2)
access type 1.3.1(2/5), 3.10(1)
  subpool 13.11.4(22/3)
access types
  input-output unspecified A.7(6/6)
access value 3.10(1)
access-to-constant type 3.10(10)
access-to-object type 3.10(7/1)
access-to-subprogram type 3.10(7/1), 3.10(11)
access-to-variable type 3.10(10)
Access_Check 11.5(11/2)
  [*partial*] 4.1(13), 4.1.5(8/3), 4.6(51/5), 4.8(10.4/3)
access_definition 3.10(6/2)
  used 3.3.1(2/3), 3.6(7/2), 3.7(5/5), 5.5.2(2.1/5), 6.1(13/2), 6.1(15/5), 6.5(2.3/2), 8.5.1(2/5), 12.4(2/3), P.1
access_to_object_definition 3.10(3)
  used 3.10(2/2), P.1
access_to_subprogram_definition 3.10(5)
  used 3.10(2/2), P.1
access_type_definition 3.10(2/2)
  used 3.2.1(4/2), 12.5.4(2), P.1
accessibility
  distributed 3.10.2(32.1/3)
  from shared passive library units E.2.1(8)
accessibility level 1.3.1(4/5), 3.10.2(3/5)
accessibility rule
  Access attribute 3.10.2(28/3), 3.10.2(32/5)
  requeue statement 9.5.4(6/3)
  type conversion 4.6(24.14/6), 4.6(24.21/6), 4.6(24.25/6)
  type conversion, array components 4.6(24.6/2)
Accessibility_Check 11.5(19.1/2)
  [*partial*] 3.10.2(29), 4.6(39.1/2), 4.6(48/3), 4.8(10.1/6), 6.5(8/5), 6.5(21/3), 9.5.4(7.2/5), 13.11.4(25/3), 13.11.4(26/3), E.4(18/1)
accessible partition E.1(7)
accumulator
  reduction attribute 4.5.10(24/6)
accuracy 4.6(32), G.2(1)
ACK
  in Ada.Characters.Latin_1 A.3.3(5)
acquire
  execution resource associated with protected object 9.5.1(5/4)
actions
  concurrent 9.10(16/5)
  conflicting 9.10(17/5)
  potentially conflicting 9.10(17/5)
  sequential 9.10(11/5)
activation
  of a task 9.2(1)
activation failure 9.2(1)
Activation_Is_Complete
  in Ada.Task_Identification C.7.1(4/3)
activator
  of a task 9.2(5)
active
  absolute deadline D.2.6(14/5)
active locale A.19(8/3)

active partition 10.2(28/3), E.1(2)
active priority D.1(15/5)
actual 12.3(7/5)
actual duration D.9(12/5)
actual parameter
  for a formal parameter 6.4.1(3)
actual subtype 3.3(23/5), 12.5(4)
  of an object 3.3.1(9/2)
actual type 12.5(4)
actual_parameter_part 6.4(4)
  used 4.1.6(10/3), 6.4(2), 6.4(3), 9.5.3(2), P.1
Actual_Quantum
  in Ada.Dispatching.Round_Robin D.2.5(4/5)
Acute
  in Ada.Characters.Latin_1 A.3.3(22)
Ada A.2(2/5)
Ada calling convention 6.3.1(3/3)
Ada.Ada.Unchecked_Deallocate_Subpool 13.11.5(3/6)
Ada.Assertions 11.4.2(12/5)
Ada.Asynchronous_Task_Control D.11(3/5)
Ada.Calendar 9.6(10/5)
Ada.Calendar.Arithmetic 9.6.1(8/5)
Ada.Calendar.Formatting 9.6.1(15/5)
Ada.Calendar.Time_Zones 9.6.1(2/5)
Ada.Characters A.3.1(2/5)
Ada.Characters.Conversions A.3.4(2/5)
Ada.Characters.Handling A.3.2(2/5)
Ada.Characters.Latin_1 A.3.3(3/5)
Ada.Command_Line A.15(3/5)
Ada.Complex_Text_IO G.1.3(9.1/2)
Ada.Containers A.18.1(3/5)
Ada.Containers.Bounded_Doubly_Linked_Lists A.18.20(1/3)
Ada.Containers.Bounded_Hashed_Maps A.18.21(1/3)
Ada.Containers.Bounded_Hashed_Sets A.18.23(1/3)
Ada.Containers.Bounded_Indefinite_Holders A.18.32(1/5)
Ada.Containers.Bounded_Multiway_Trees A.18.25(1/3)
Ada.Containers.Bounded_Ordered_Maps A.18.22(1/3)
Ada.Containers.Bounded_Ordered_Sets A.18.24(1/3)
Ada.Containers.Bounded_Priority_Queues A.18.31(2/6)
Ada.Containers.Bounded_Synchronized_Queues A.18.29(2/6)
Ada.Containers.Bounded_Vectors A.18.19(1/3)
Ada.Containers.Doubly_Linked_Lists A.18.3(5/5)
Ada.Containers.Generic_Array_Sort A.18.26(3/5)
Ada.Containers.Generic_Constrained_Array_Sort A.18.26(7/5)
Ada.Containers.Generic_Sort A.18.26(9.2/5)
Ada.Containers.Hashed_Maps A.18.5(2/5)
Ada.Containers.Hashed_Sets A.18.8(2/5)
Ada.Containers.Indefinite_Doubly_Linked_Lists A.18.12(2/3)
Ada.Containers.Indefinite_Hashed_Maps A.18.13(2/3)
Ada.Containers.Indefinite_Hashed_Sets A.18.15(2/3)
Ada.Containers.Indefinite_Holders A.18.18(5/5)
Ada.Containers.Indefinite_Multiway_Trees A.18.17(2/3)
Ada.Containers.Indefinite_Ordered_Maps A.18.14(2/3)
Ada.Containers.Indefinite_Ordered_Sets A.18.16(2/3)
Ada.Containers.Indefinite_Vectors A.18.11(2/3)
Ada.Containers.Multiway_Trees A.18.10(7/5)
Ada.Containers.Ordered_Maps A.18.6(2/5)
Ada.Containers.Ordered_Sets A.18.9(2/5)
Ada.Containers.Synchronized_Queue_Interfaces A.18.27(3/5)
Ada.Containers.Unbounded_Priority_Queues A.18.30(2/5)
Ada.Containers.Unbounded_Synchronized_Queues A.18.28(2/5)
Ada.Containers.Vectors A.18.2(6/5)
Ada.Decimal F.2(2/5)
Ada.Direct_IO A.8.4(2/5)
Ada.Directories A.16(3/5)
Ada.Directories.Hierarchical_File_Names A.16.1(3/5)
Ada.Directories.Information A.16(124/2)
Ada.Dispatching D.2.1(1.2/5)
Ada.Dispatching.EDF D.2.6(9/5)
Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)
Ada.Dispatching.Round_Robin D.2.5(4/5)
Ada.Dynamic_Priorities D.5.1(3/5)
Ada.Environment_Variables A.17(3/5)
Ada.Exceptions 11.4.1(2/5)
Ada.Execution_Time D.14(3/5)
Ada.Execution_Time.Group_Budgets D.14.2(3/5)
Ada.Execution_Time.Interrupts D.14.3(3/5)
Ada.Execution_Time.Timers D.14.1(3/5)
Ada.Finalization 7.6(4/5)
Ada.Float_Text_IO A.10.9(33)
Ada.Float_Wide_Text_IO A.11(2/2)
Ada.Float_Wide_Wide_Text_IO A.11(3/2)
Ada.Integer_Text_IO A.10.8(21)
Ada.Integer_Wide_Text_IO A.11(2/2)
Ada.Integer_Wide_Wide_Text_IO A.11(3/2)
Ada.Interrupts C.3.2(2/5)
Ada.Interrupts.Names C.3.2(12/5)
Ada.IO_Exceptions A.13(3/5)
Ada.Iterator_Interfaces 5.5.1(2/5)
Ada.Locales A.19(3/5)
Ada.Numerics A.5(3/5)
Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5), A.5.6(2/5)
Ada.Numerics.Big_Numbers.Big_Reals A.5.7(2/5)
Ada.Numerics.Complex_Arrays G.3.2(53/2)
Ada.Numerics.Complex_Elementary_Functions G.1.2(9/1)
Ada.Numerics.Complex_Types G.1.1(25/1)
Ada.Numerics.Discrete_Random A.5.2(17/5)
Ada.Numerics.Elementary_Functions A.5.1(9/1)
Ada.Numerics.Float_Random A.5.2(5/5)
Ada.Numerics.Generic_Complex_Arrays G.3.2(2/5)
Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(2/5)
Ada.Numerics.Generic_Complex_Types G.1.1(2/5)
Ada.Numerics.Generic_Elementary_Functions A.5.1(3/5)
Ada.Numerics.Generic_Real_Arrays G.3.1(2/5)
Ada.Numerics.Real_Arrays G.3.1(31/2)
Ada.Real_Time D.8(3/5)
Ada.Real_Time.Timing_Events D.15(3/5)
Ada.Sequential_IO A.8.1(2/5)
Ada.Storage_IO A.9(3/5)
Ada.Streams 13.13.1(2/5)
Ada.Streams.Storage 13.13.1(12/5)
Ada.Streams.Storage.Bounded 13.13.1(26/5)
Ada.Streams.Storage.Unbounded 13.13.1(18/5)
Ada.Streams.Stream_IO A.12.1(3/5)
Ada.Strings A.4.1(3/5)
Ada.Strings.Bounded A.4.4(3/5)
Ada.Strings.Bounded.Equal_Case_Insensitive A.4.10(7/5)
Ada.Strings.Bounded.Hash A.4.9(7/5)
Ada.Strings.Bounded.Hash_Case_Insensitive A.4.9(11.7/5)
Ada.Strings.Bounded.Less_Case_Insensitive A.4.10(18/5)

Ada.Strings.Equal_Case_Insensitive A.4.10(2/5)
Ada.Strings.Fixed A.4.3(5/5)
Ada.Strings.Fixed.Equal_Case_Insensitive A.4.10(5/3)
Ada.Strings.Fixed.Hash_Case_Insensitive A.4.9(11.5/3)
Ada.Strings.Fixed.Less_Case_Insensitive A.4.10(16/3)
Ada.Strings.Hash A.4.9(2/5)
Ada.Strings.Hash_Case_Insensitive A.4.9(11.2/5)
Ada.Strings.Less_Case_Insensitive A.4.10(13/5)
Ada.Strings.Maps A.4.2(3/5)
Ada.Strings.Maps.Constants A.4.6(3/5)
Ada.Strings.Text_Buffers A.4.12(3/5)
Ada.Strings.Text_Buffers.Bounded A.4.12(26/5)
Ada.Strings.Text_Buffers.Unbounded A.4.12(18/5)
Ada.Strings.Unbounded A.4.5(3/5)
Ada.Strings.Unbounded.Equal_Case_Insensitive A.4.10(10/5)
Ada.Strings.Unbounded.Hash A.4.9(10/5)
Ada.Strings.Unbounded.Hash_Case_Insensitive A.4.9(11.10/5)
Ada.Strings.Unbounded.Less_Case_Insensitive A.4.10(21/5)
Ada.Strings.UTF_Encoding A.4.11(3/5)
Ada.Strings.UTF_Encoding.Conversions A.4.11(15/5)
Ada.Strings.UTF_Encoding.Strings A.4.11(22/5)
Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(30/5)
Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(38/5)
Ada.Strings.Wide_Bounded A.4.7(1/3)
Ada.Strings.Wide_Bounded.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Bounded.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Bounded.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Fixed A.4.7(1/3)
Ada.Strings.Wide_Fixed.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Fixed.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Fixed.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Maps A.4.7(3/5)
Ada.Strings.Wide_Maps.Wide_Constants A.4.7(1/3), A.4.8(28/2)
Ada.Strings.Wide_Unbounded A.4.7(1/3)
Ada.Strings.Wide_Unbounded.Wide_Equal_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Unbounded.Wide_Hash A.4.7(1/3)
Ada.Strings.Wide_Unbounded.Wide_Hash_Case_Insensitive A.4.7(1/3)
Ada.Strings.Wide_Wide_Bounded A.4.8(1/3)
Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Bounded.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Fixed.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Maps A.4.8(3/5)
Ada.Strings.Wide_Wide_Maps.Wide_Wide_Constants A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Equal_Case_Insensitive A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Hash A.4.8(1/3)
Ada.Strings.Wide_Wide_Unbounded.Wide_Wide_Hash_Case_Insensitive A.4.8(1/3)
Ada.Synchronous_Barriers D.10.1(3/5)
Ada.Synchronous_Task_Control D.10(3/5)
Ada.Synchronous_Task_Control.EDF D.10(5.2/5)
Ada.Tags 3.9(6/5)
Ada.Tags.Generic_Dispatching_Constructor 3.9(18.2/5)
Ada.Task_Attributes C.7.2(2/5)
Ada.Task_Identification C.7.1(2/5)
Ada.Task_Termination C.7.3(2/5)
Ada.Text_IO A.10.1(2/5)
Ada.Text_IO.Bounded_IO A.10.11(3/5)
Ada.Text_IO.Complex_IO G.1.3(3/5)
Ada.Text_IO.Editing F.3.3(3/5)
Ada.Text_IO.Text_Streams A.12.2(3/5)
Ada.Text_IO.Unbounded_IO A.10.12(3/5)
Ada.Unchecked_Conversion 13.9(3/5)
Ada.Unchecked_Deallocate_Subpool *child of* Ada 13.11.5(3/6)
Ada.Unchecked_Deallocation 13.11.2(3/5)
Ada.Wide_Characters A.3.1(4/5)
Ada.Wide_Characters.Handling A.3.5(3/5)
Ada.Wide_Command_Line A.15.1(2/5)
Ada.Wide_Directories A.16.2(2/5)
Ada.Wide_Directories.Hierarchical_File_Names A.16.2(2/5)
Ada.Wide_Directories.Information A.16.2(2/5)
Ada.Wide_Environment_Variables A.17.1(2/5)
Ada.Wide_Text_IO A.11(2/2)
Ada.Wide_Text_IO.Complex_IO G.1.4(1)
Ada.Wide_Text_IO.Editing F.3.4(1)
Ada.Wide_Text_IO.Text_Streams A.12.3(3/5)
Ada.Wide_Text_IO.Wide_Bounded_IO A.11(4/3)
Ada.Wide_Text_IO.Wide_Unbounded_IO A.11(5/3)
Ada.Wide_Wide_Characters A.3.1(6/5)
Ada.Wide_Wide_Characters.Handling A.3.6(1/3)
Ada.Wide_Wide_Command_Line A.15.1(3/5)
Ada.Wide_Wide_Directories A.16.2(3/5)
Ada.Wide_Wide_Directories.Hierarchical_File_Names A.16.2(3/5)
Ada.Wide_Wide_Directories.Information A.16.2(3/5)
Ada.Wide_Wide_Environment_Variables A.17.1(3/5)
Ada.Wide_Wide_Text_IO A.11(3/2)
Ada.Wide_Wide_Text_IO.Complex_IO G.1.5(1/2)
Ada.Wide_Wide_Text_IO.Editing F.3.5(1/2)
Ada.Wide_Wide_Text_IO.Text_Streams A.12.4(3/5)

Ada.Wide_Wide_-
Text_IO.Wide_Wide_Bounded_IO
A.11(4/3)
Ada.Wide_Wide_-
Text_IO.Wide_Wide_Unbounded_IO
A.11(5/3)
Ada_To_COBOL
in Interfaces.COBOL B.4(14)
adafinal B.1(39/3)
adainit B.1(39/3)
Add
in
Ada.Execution_Time.Group_Budgets
D.14.2(9/2)
Add_Task
in
Ada.Execution_Time.Group_Budgets
D.14.2(8/2)
additive
aspect 13.1.1(18.3/5)
address
arithmetic 13.7.1(6)
comparison 13.7(14/3)
in System 13.7(12)
Address aspect 13.3(12)
Address attribute 13.3(11), J.7.1(5)
Address clause 13.3(7/2), 13.3(12)
Address_To_Access_Conversions
child of System 13.7.2(2/5)
Adjacent attribute A.5.3(48)
Adjust 7.6(2)
in Ada.Finalization 7.6(6/2)
adjusting the value of an object 7.6(15),
7.6(16/3)
adjustment 7.6(15), 7.6(16/3)
as part of assignment 5.2(14/5)
Admission_Policy pragma D.4.1(3/5),
L(1.1/5)
ADT (abstract data type)
See private types and private
extensions 7.3(1)
See also abstract type 3.9.3(1/2)
advice 1.1.2(37)
Aft attribute 3.5.10(5)
aggregate 1.3.3(1/5), 4.3(1), 4.3(2/5)
index parameter 4.3.3(6.1/5)
used 4.4(7/5), 4.7(2), 6.8(2/4),
13.1.1(4/5), P.1
See also composite type 3.2(2/5)
Aggregate aspect 4.3.5(3/5)
expected type has 4.3.5(22/6)
aliased 3.10(9/5)
aliased view 1.3.1(5/5)
aliasing
See distinct access paths 6.2(12/3)
Alignment
in Ada.Strings A.4.1(6)
Alignment (subtype) aspect 13.3(26.4/2)
Alignment attribute 13.3(23/2),
13.3(26.2/2)
Alignment clause 13.3(7/2), 13.3(25/2),
13.3(26.4/2)
All_Calls_Remote aspect E.2.3(16/5)
All_Calls_Remote pragma J.15.15(6/5),
L(2.1/5)
All_Checks 11.5(25/3)
All_Conflict_Checks conflict check
policy 9.10.1(22/5)
All_Parallel_Conflict_Checks conflict
check policy 9.10.1(20/5)
All_Tasking_Conflict_Checks conflict
check policy 9.10.1(21/5)
Allocate
in System.Storage_Pools 13.11(7)
in System.Storage_Pools.Subpools
13.11.4(14/3)
Allocate_From_Subpool
in System.Storage_Pools.Subpools
13.11.4(11/5)
Allocation_Check 11.5(19.2/2)
[*partial*] 4.8(10.2/2), 4.8(10.3/2),
4.8(10.4/3), 13.11.4(30/3)
allocator 4.8(2/3)
used 4.4(7/5), P.1
allows blocking 9.5(21/5)
allows exit 5.5.3(19/5)
Alphanumeric
in Interfaces.COBOL B.4(16/3)
alphanumeric character
a category of Character A.3.2(31)
Alphanumeric_Set
in Ada.Strings.Maps.Constants
A.4.6(4)
ambiguous 8.6(30)
ambiguous cursor
of a vector A.18.2(240/2)
ampersand 2.1(15/6)
in Ada.Characters.Latin_1 A.3.3(8)
ampersand operator 4.5(1), 4.5.3(3)
ancestor
of a library unit 10.1.1(11)
of a tree node A.18.10(4/3)
of a type 3.4.1(10/2)
ultimate 3.4.1(10/2)
ancestor of a type 1.3.1(7/5)
ancestor subtype
of a formal derived type 12.5.1(5/3)
of a private_extension_declaration
7.3(8)
ancestor type
of an extension_aggregate 4.3.2(5/3)
Ancestor_Find
in Ada.Containers.Multiway_Trees
A.18.10(40.1/5), A.18.10(40/5)
ancestor_part 4.3.2(3)
used 4.3.2(2), P.1
and operator 4.5(1), 4.5.1(2)
and then (short-circuit control form)
4.5(1), 4.5.1(1)
angle threshold G.2.4(10)
Annex
informative 1.1.2(18)
normative 1.1.2(14)
Specialized Needs 1.1.2(7)
Annotated Ada Reference Manual
0.2(5/3)
anonymous access type 3.10(12/3)
anonymous allocator 3.10.2(14/6)
anonymous array type 3.3.1(1/3)
anonymous protected type 3.3.1(1/3)
anonymous task type 3.3.1(1/3)
anonymous type 3.2.1(7/2)
Any_Priority *subtype of* Integer
in System 13.7(16)
APC
in Ada.Characters.Latin_1 A.3.3(19)
apostrophe 2.1(15/6)
in Ada.Characters.Latin_1 A.3.3(8)
Append
in Ada.Containers.Doubly_Linked_-
Lists A.18.3(23.1/5), A.18.3(23/5)
in Ada.Containers.Vectors
A.18.2(47.1/5), A.18.2(47/5)
in Ada.Strings.Bounded A.4.4(13),
A.4.4(14), A.4.4(15), A.4.4(16),
A.4.4(17), A.4.4(18), A.4.4(19),
A.4.4(20)
in Ada.Strings.Unbounded A.4.5(12),
A.4.5(13), A.4.5(14)
Append_Child
in Ada.Containers.Multiway_Trees
A.18.10(52/5)
Append_Vector
in Ada.Containers.Vectors
A.18.2(46/5)
applicable index constraint 4.3.3(10/5)
application areas 1.1.2(7)
applies
aspect 13.1.1(23/3), 13.1.1(27/3),
13.1.1(29/3), 13.1.1(30/3)
apply
to a callable construct by a return
statement 6.5(4/2)
to a loop_statement by an
exit_statement 5.7(4)
to a program unit by a program unit
pragma J.15(2/5)
arbitrary order 1.1.4(18)
allowed 2.8(12/6), 3.3.1(20/2), 3.5(9),
3.6(22/2), 3.11(10/1), 3.11(11/3),
3.11(13), 4.1.1(7), 4.1.2(7), 4.3(5),
4.3.1(19/5), 4.3.2(7), 4.3.3(20.3/6),
4.3.3(20.4/6), 4.3.3(22), 4.3.3(23),
4.3.5(43/5), 4.3.5(44/6), 4.3.5(45/5),
4.3.5(46/6), 4.5.2(27/4), 4.5.10(24/6),
4.8(10/2), 5.2(7), 6.1.1(34/5),
6.1.1(35/3), 6.4(10/2), 6.4.1(17),
7.6(12), 7.6(16/3), 7.6.1(9/3),
7.6.1(11.1/3), 7.6.1(20.2/3), 9.7.1(15),
9.8(4), 12.3(20), 13.11.5(7/3)

Arccos
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(6)
Arccosh
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
Arccot
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(6)
Arccoth
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
Arcsin
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(6)
Arcsinh
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
Arctan
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(5)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(6)
Arctanh
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(7)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
Argument
  in Ada.Command_Line A.15(5)
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(10/2), G.3.2(31/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(10)
argument of a pragma 2.8(9)
Argument_Count
  in Ada.Command_Line A.15(4)
Argument_Error
  in Ada.Numerics A.5(3/5)
Arithmetic
  child of Ada.Calendar 9.6.1(8/5)
array 3.6(1)
array component expression 4.3.3(6)
array component iterator 5.5.2(3/3)
array for a loop 5.5.2(11/6)
array indexing
  See indexed_component 4.1.1(1)
array slice 4.1.2(1)
array type 1.3.1(9/5), 3.6(1)
array_aggregate 4.3.3(2/5)
  used 4.3.3(2/5), 13.4(3), P.1
array_component_association 4.3.3(5/5)
  used 4.3.3(4.1/5), P.1
array_component_association_list 4.3.3(4.1/5)
  used 4.3.3(4/5), 4.3.4(4/5), P.1
array_delta_aggregate 4.3.4(4/5)
  used 4.3.4(2/5), P.1
array_type_definition 3.6(2)
  used 3.2.1(4/2), 3.3.1(2/3), 12.5.3(2), P.1
ASCII
  package physically nested within the declaration of Standard A.1(36.3/2)
  in Standard A.1(36.3/2)
aspect 1.3.1(10/5), 13.1(0.1/6), K.1(1/3)
  additive 13.1.1(18.3/5)
  assertion 11.4.2(9/5)
  class-wide 13.1.1(28/4)
  implicitly composed 13.1.1(18.3/5)
  interfacing B.1(0.1/3)
  library unit 13.1.1(32/6)
  nonoverridable 13.1.1(18.3/5)
  predicate 3.2.4(1/5)
  user-defined literal 4.2.1(2/5)
aspect of representation 13.1(8/5)
aspect-related pragma [*partial*] 2.8(7.1/6)
  Attach_Handler J.15.7(3/3)
  CPU J.15.9(1/3)
  Dispatching_Domain J.15.10(1/3)
  Interrupt_Handler J.15.7(1/3)
  Interrupt_Priority J.15.11(3/3)
  Preelaborable_Initialization J.15.14(7/5)
  Priority J.15.11(1/3)
  Relative_Deadline J.15.12(1/3)
  Storage_Size J.15.4(1/3)
  See also representation pragma 13.1(4/6)
aspect_clause 13.1(2/1)
  used 3.8(5/1), 3.11(4/1), 9.1(5/1), 9.4(5/1), 9.4(8/4), P.1
aspect_definition 13.1.1(4/5)
  used 13.1.1(2/3), P.1
aspect_mark 13.1.1(3/3)
  used 2.8(3/3), 11.4.2(6.1/3), 13.1.1(2/3), L(2.4/3), P.1
aspect_specification 13.1.1(2/3)
  used 3.2.1(3/3), 3.2.2(2/3), 3.3.1(2/3), 3.7(5/5), 3.8(6/3), 3.9.3(1.1/3), 4.5.10(3/5), 5.5(3/5), 5.6.1(2/5), 6.1(2/3), 6.1(15/5), 6.3(2/3), 6.5(2.1/5), 6.7(2/3), 6.8(2/4), 7.1(3/3), 7.2(2/3), 7.3(2/3), 7.3(3/3), 8.5.1(2/5), 8.5.2(2/3), 8.5.3(2/3), 8.5.4(2/3), 8.5.5(2/3), 9.1(2/3), 9.1(3/5), 9.1(6/3), 9.4(2/3), 9.4(3/3), 9.4(7/3), 9.5.2(2/3), 9.5.2(5/5), 9.5.2(8/5), 10.1.3(3/3), 10.1.3(4/3), 10.1.3(5/3), 10.1.3(6/3), 11.1(2/3), 12.1(3/3), 12.3(2/3), 12.4(2/3), 12.5(2.1/5), 12.6(2.1/3), 12.6(2.2/3), 12.7(2/3), P.1
aspects
  Address 13.3(12)
  Aggregate 4.3.5(3/5)
  Alignment (subtype) 13.3(26.4/2)
  All_Calls_Remote E.2.3(16/5)
  Asynchronous E.4.1(8.1/3)
  Atomic C.6(6.2/3)
  Atomic_Components C.6(6.7/3)
  Attach_Handler C.3.1(6.3/3)
  Bit_Order 13.5.3(4)
  Coding 13.4(7)
  Component_Size 13.3(70)
  Constant_Indexing 4.1.6(2/5)
  Constructor_Extension 3.9.1(4.4/6)
  Convention B.1(2/3)
  CPU D.16(8/3)
  Default_Component_Value 3.6(22.2/3)
  Default_Initial_Condition 7.3.3(2/5)
  Default_Iterator 5.5.1(8/5)
  Default_Storage_Pool 13.11.3(5/4)
  Default_Value 3.5(56.3/3)
  Discard_Names C.5(1.3/4)
  Dispatching H.7.1(8/5)
  Dispatching_Domain D.16.1(18/3)
  Dynamic_Predicate 3.2.4(1/5)
  Elaborate_Body 10.2.1(26.1/5)
  Exclusive_Functions 9.5.1(2.2/4)
  Export B.1(1/3)
  External_Name B.1(1/3)
  External_Tag 13.3(75/3), K.2(65)
  Full_Access_Only C.6(6.5/5)
  Global 6.1.2(11/5)
  Global'Class 6.1.2(15/5)
  Implicit_Dereference 4.1.5(2/3)
  Import B.1(1/3)
  Independent C.6(6.3/3)
  Independent_Components C.6(6.10/3)
  Inline 6.3.2(5.1/3)
  Input 13.13.2(38/5)
  Input'Class 13.13.2(38/5)
  Integer_Literal 4.2.1(3/5)
  Interrupt_Handler C.3.1(6.2/3)
  Interrupt_Priority D.1(6.3/3)
  Iterator_Element 5.5.1(9/5)
  Iterator_View 5.5.1(9.1/5)
  Layout 13.5(1)

Link_Name B.1(1/3)
Machine_Radix F.1(1/5)
Max_Entry_Queue_Length D.4(18/5)
No_Controlled_Parts H.4.1(2/5)
No_Return 6.5.1(3.2/3)
Nonblocking 9.5(19/5)
Output 13.13.2(38/5)
Output'Class 13.13.2(38/5)
Pack 13.2(5.1/3)
Parallel_Calls 9.10.1(24/5)
Post 6.1.1(4/5)
Post'Class 6.1.1(5/5)
Pre 6.1.1(2/5)
Pre'Class 6.1.1(3/5)
Predicate_Failure 3.2.4(14.2/4)
Preelaborate 10.2.1(11/5)
Priority D.1(6.2/3)
Put_Image 4.10(4/6)
Read 13.13.2(38/5)
Read'Class 13.13.2(38/5)
Real_Literal 4.2.1(4/5)
Record layout 13.5(1)
Relative_Deadline D.2.6(9.2/3)
Remote_Call_Interface E.2.3(7/5)
Remote_Types E.2.2(4/5)
Shared_Passive E.2.1(4/5)
Size (object) 13.3(41)
Size (subtype) 13.3(48)
Small 3.5.10(2/5)
Stable_Properties 7.3.4(5/5)
Stable_Properties'Class 7.3.4(6/5)
Static 6.8(3.6/5)
Static_Predicate 3.2.4(1/5)
Storage_Pool 13.11(15/5)
Storage_Size (access) 13.11(15/5)
Storage_Size (task) 13.3(65.2/3)
Stream_Size 13.13.2(1.5/2)
String_Literal 4.2.1(5/5)
Synchronization 9.5(12/3)
Type_Invariant 7.3.2(2/3)
Type_Invariant'Class 7.3.2(3/5)
Unchecked_Union B.3.3(3.2/3)
Use_Formal H.7.1(3/5)
Variable_Indexing 4.1.6(3/5)
Volatile C.6(6.4/3)
Volatile_Components C.6(6.8/3)
Write 13.13.2(38/5)
Write'Class 13.13.2(38/5)
Yield D.2.1(1.7/5)
assembly language C.1(4/3)
Assert
in Ada.Assertions 11.4.2(14/2)
Assert pragma 11.4.2(3/2), L(2.2/2)
assertion 1.3.4(1/5)
assertion aspect 11.4.2(9/5)
assertion expressions 11.4.2(1.1/5)
assertion policy
Assert pragma 11.4.2(18/3)
Assertion_Error
raised by failure of assertion 11.4.2(18/3)
raised by failure of runtime check 3.2.4(31.1/4), 4.6(57/5), 6.1.1(32/3), 6.1.1(33/3), 6.1.1(35/3), 6.5(22/5), 7.3.2(22/3), 7.3.3(9/5), 11.5(23.3/5)
in Ada.Assertions 11.4.2(13/2)
Assertion_Policy pragma 11.4.2(6.1/3), 11.4.2(6/2), L(2.3/2), L(2.4/3)
assertions 11.4.2(1.1/5)
child of Ada 11.4.2(12/5)
Assign
in Ada.Containers.Doubly_Linked_-Lists A.18.3(17.5/5), A.18.3(51.8/5)
in Ada.Containers.Hashed_Maps A.18.5(17.7/5), A.18.5(37.9/5)
in Ada.Containers.Hashed_Sets A.18.8(17.4/5), A.18.8(59.8/5)
in Ada.Containers.Indefinite_Holders A.18.18(20/5)
in Ada.Containers.Multiway_Trees A.18.10(32/5), A.18.10(70.8/5)
in Ada.Containers.Ordered_Maps A.18.6(16.7/5), A.18.6(51.10/5)
in Ada.Containers.Ordered_Sets A.18.9(16.4/5), A.18.9(74.8/5)
in Ada.Containers.Vectors A.18.2(34.7/5), A.18.2(79.8/5)
See assignment operation 5.2(3)
Assign_Task
in System.Multiprocessors.Dispatching_Domains D.16.1(11/3)
assigning back of parameters 6.4.1(17)
assignment
user-defined 7.6(1)
assignment operation 5.2(3), 5.2(12/5), 7.6(13)
during elaboration of an object_declaration 3.3.1(18/2)
during evaluation of a generic_association for a formal object of mode in 12.4(11)
during evaluation of a parameter_association 6.4.1(11)
during evaluation of an aggregate 4.3(5)
during evaluation of an initialized allocator 4.8(7/2)
during evaluation of an uninitialized allocator 4.8(9/2)
during evaluation of concatenation 4.5.3(10)
during execution of a for loop 5.5(9/5)
during execution of an assignment_statement 5.2(12/5)
during parameter copy back 6.4.1(17)
assignment_statement 5.2(2)
used 5.1(4/2), P.1
associated components
of a record_component_association 4.3.1(10)
associated declaration
of an aspect specification 13.1.1(1/3)
associated discriminants
of a named discriminant_association 3.7.1(5)
of a positional discriminant_association 3.7.1(5)
associated entity
of an aspect specification 13.1.1(5/5)
associated object
of a value of a by-reference type 6.2(10/5)
asterisk 2.1(15/6)
in Ada.Characters.Latin_1 A.3.3(8)
asynchronous
remote procedure call E.4.1(9/3)
Asynchronous aspect E.4.1(8.1/3)
Asynchronous pragma J.15.13(2/3), L(3.1/3)
asynchronous remote procedure call E.4(1/5)
asynchronous_select 9.7.4(2)
used 9.7(2), P.1
Asynchronous_Task_Control
child of Ada D.11(3/5)
at sign 2.1(15/6)
at-most-once execution E.4(11)
at_clause J.7(1)
used 13.1(2/1), P.1
atomic C.6(7/3)
Atomic aspect C.6(6.2/3)
Atomic pragma J.15.8(2/3), L(4.1/3)
Atomic_Add
in System.Atomic_Operations.Integer_Arithmetic C.6.4(4/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(4/5)
Atomic_Clear
in System.Atomic_Operations.Test_And_Set C.6.3(6/5)
Atomic_Compare_And_Exchange
in System.Atomic_Operations.Exchange C.6.2(7/5)
Atomic_Components aspect C.6(6.7/3)
Atomic_Components pragma J.15.8(5/3), L(5.1/3)
Atomic_Exchange
in System.Atomic_Operations.Exchange C.6.2(6/5)

Atomic_Fetch_And_Add
  in System.Atomic_Operations.Integer_Arithmetic C.6.4(6/5)
  in System.Atomic_Operations.Modular_Arithmetic C.6.5(6/5)
Atomic_Fetch_And_Subtract
  in System.Atomic_Operations.Integer_Arithmetic C.6.4(7/5)
  in System.Atomic_Operations.Modular_Arithmetic C.6.5(7/5)
Atomic_Operations
  child of System C.6.1(3/5)
Atomic_Subtract
  in System.Atomic_Operations.Integer_Arithmetic C.6.4(5/5)
  in System.Atomic_Operations.Modular_Arithmetic C.6.5(5/5)
Atomic_Test_And_Set
  in System.Atomic_Operations.Test_And_Set C.6.3(5/5)
Attach_Handler
  in Ada.Interrupts C.3.2(7)
Attach_Handler aspect C.3.1(6.3/3)
Attach_Handler pragma J.15.7(4/3), L(6.1/3)
attaching
  to an interrupt C.3(2)
attribute 1.3.1(12/5), 4.1.4(1), K.2(1/3)
  representation 13.3(1/1)
  specifiable 13.3(5/3)
  specifying 13.3(1/1)
  stream-oriented 13.13.2(1/3)
attribute_definition_clause 13.3(2)
  used 13.1(2/1), P.1
attribute_designator 4.1.4(3/2)
  used 4.1.4(2/5), 13.1(3), 13.3(2), P.1
Attribute_Handle
  in Ada.Task_Attributes C.7.2(3)
attribute_reference 4.1.4(2/5)
  used 4.1(2/5), P.1
attributes
  Access 3.10.2(24/1), 3.10.2(32/5)
  Address 13.3(11), J.7.1(5)
  Adjacent A.5.3(48)
  Aft 3.5.10(5)
  Alignment 13.3(23/2), 13.3(26.2/2)
  Base 3.5(15)
  Bit_Order 13.5.3(4)
  Body_Version E.3(4)
  Callable 9.9(2)
  Caller C.7.1(14/3)
  Ceiling A.5.3(33), A.5.4(7/6)
  Class 3.9(14), 7.3.1(9), J.11(2/2)
  Component_Size 13.3(69)
  Compose A.5.3(24)
  Constrained 3.7.2(3/5), J.4(2)
  Copy_Sign A.5.3(51)
  Count 9.9(5)
  Definite 12.5.1(23/3)
  Delta 3.5.10(3)
  Denorm A.5.3(9)
  Digits 3.5.8(2/1), 3.5.10(7)
  Enum_Rep 13.4(10.2/5)
  Enum_Val 13.4(10.5/5)
  Exponent A.5.3(18)
  External_Tag 13.3(75/3)
  First 3.5(12), 3.6.2(3)
  First(N) 3.6.2(4)
  First_Bit 13.5.2(3/2)
  First_Valid 3.5.5(7.2/4)
  Floor A.5.3(30), A.5.4(8/6)
  Fore 3.5.10(4)
  Fraction A.5.3(21)
  Has_Same_Storage 13.3(73.2/4)
  Identity 11.4.1(9), C.7.1(12)
  Image 4.10(34/5), 4.10(40/5)
  Index 6.1.1(30.2/5)
  Input 13.13.2(22), 13.13.2(32)
  Last 3.5(13), 3.6.2(5)
  Last(N) 3.6.2(6)
  Last_Bit 13.5.2(4/2)
  Last_Valid 3.5.5(7.3/4)
  Leading_Part A.5.3(54)
  Length 3.6.2(9)
  Length(N) 3.6.2(10)
  Machine A.5.3(60)
  Machine_Emax A.5.3(8)
  Machine_Emin A.5.3(7)
  Machine_Mantissa A.5.3(6)
  Machine_Overflows A.5.3(12), A.5.4(4)
  Machine_Radix A.5.3(2), A.5.4(2)
  Machine_Rounding A.5.3(41.1/2), A.5.4(11/6)
  Machine_Rounds A.5.3(11), A.5.4(3)
  Max 3.5(19)
  Max_Alignment_For_Allocation 13.11.1(4/5)
  Max_Size_In_Storage_Elements 13.11.1(3/5)
  Min 3.5(16)
  Mod 3.5.4(16.1/2)
  Model A.5.3(68), G.2.2(7)
  Model_Emin A.5.3(65), G.2.2(4)
  Model_Epsilon A.5.3(66)
  Model_Mantissa A.5.3(64), G.2.2(3/2)
  Model_Small A.5.3(67)
  Modulus 3.5.4(17)
  Object_Size 13.3(58.2/5)
  Old 6.1.1(26/5)
  Output 13.13.2(19), 13.13.2(29)
  Overlaps_Storage 13.3(73.6/3)
  Parallel_Reduce(Reducer, Initial_Value) 4.5.10(33/5)
  Partition_Id E.1(9)
  Pos 3.5.5(2)
  Position 13.5.2(2/2)
  Pred 3.5(25)
  Preelaborable_Initialization 10.2.1(11.10/6)
  Priority D.5.2(3/2)
  Put_Image 4.10(3/5)
  Range 3.5(14), 3.6.2(7)
  Range(N) 3.6.2(8)
  Read 13.13.2(6), 13.13.2(14)
  Reduce(Reducer, Initial_Value) 4.5.10(23/5), 4.5.10(32/5)
  Relative_Deadline D.5.2(3.1/5)
  Remainder A.5.3(45)
  Result 6.1.1(29/5)
  Round 3.5.10(12)
  Rounding A.5.3(36), A.5.4(9/6)
  Safe_First A.5.3(71), G.2.2(5)
  Safe_Last A.5.3(72), G.2.2(6)
  Scale 3.5.10(11)
  Scaling A.5.3(27)
  Signed_Zeros A.5.3(13)
  Size 13.3(40), 13.3(45)
  Small 3.5.10(2/5)
  Storage_Pool 13.11(13)
  Storage_Size 13.3(60/3), 13.11(14), J.9(2)
  Stream_Size 13.13.2(1.2/3)
  Succ 3.5(22)
  Tag 3.9(16), 3.9(18)
  Terminated 9.9(3)
  Truncation A.5.3(42), A.5.4(12/6)
  Unbiased_Rounding A.5.3(39), A.5.4(10/6)
  Unchecked_Access 13.10(3), H.4(18)
  Val 3.5.5(5)
  Valid 13.9.2(3/4), H(6/5)
  Value 3.5(52)
  Version E.3(3)
  Wide_Image 4.10(31/5), 4.10(39/5)
  Wide_Value 3.5(40)
  Wide_Wide_Image 4.10(30/5), 4.10(38/5)
  Wide_Wide_Value 3.5(39.1/2)
  Wide_Wide_Width 3.5(37.1/5)
  Wide_Width 3.5(38/5)
  Width 3.5(39/5)
  Write 13.13.2(3), 13.13.2(11)
automatic constructor extension 3.9.1(4.4/6)
available
  stream attribute 13.13.2(39/2)

B

Backus-Naur Form (BNF)
  complete listing P.1(1/5)
  cross reference P.2(1/3)
  notation 1.1.4(3)
  under Syntax heading 1.1.2(25)
Barrier_Limit *subtype of* Positive
  in Ada.Synchronous_Barriers D.10.1(4/3)
base 2.4.2(3), 2.4.2(6)
  absolute deadline D.2.6(14/5)
  used 2.4.2(2), P.1
base 16 literal 2.4.2(1)
base 2 literal 2.4.2(1)
base 8 literal 2.4.2(1)
Base attribute 3.5(15)
base decimal precision
  of a floating point type 3.5.7(9)
  of a floating point type 3.5.7(10/5)
base priority D.1(15/5)
base range
  of a decimal fixed point type 3.5.9(16)
  of a fixed point type 3.5.9(12)
  of a floating point type 3.5.7(8), 3.5.7(10/5)
  of a modular type 3.5.4(10)
  of a scalar type 3.5(6)
  of a signed integer type 3.5.4(9)
  of an ordinary fixed point type 3.5.9(13)
base subtype
  of a type 3.5(15)
Base_Name
  in Ada.Directories A.16(19/5)
based_literal 2.4.2(2)
  used 2.4(2), P.1
based_numeral 2.4.2(4)
  used 2.4.2(2), P.1
basic letter
  a category of Character A.3.2(27)
basic_declaration 3.1(3/3)
  used 3.11(4/1), P.1
basic_declarative_item 3.11(4/1)
  used 3.11(3), 7.1(3/3), P.1
basic_global_mode 6.1.2(5/5)
  used 6.1.2(4/5), H.7(3/5), P.1
Basic_Map
  in Ada.Strings.Maps.Constants A.4.6(5)
Basic_Set
  in Ada.Strings.Maps.Constants A.4.6(4)
become nonlimited 7.3.1(5/1), 7.5(16)
BEL
  in Ada.Characters.Latin_1 A.3.3(5)
belong
  to a range 3.5(4)
  to a subtype 3.2(8/5)
belongs
  subpool to a pool 13.11.4(20/4)
bibliography 1.2.1(1/5)
big endian 13.5.3(2)
Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(3/5)
Big_Integers
  child of Ada.Numerics.Big_Numbers A.5.5(3/5), A.5.6(2/5)
Big_Natural *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(9/5)
Big_Positive *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(8/5)
Big_Real
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(3/5)
Big_Reals
  child of Ada.Numerics.Big_Numbers A.5.7(2/5)
binary
  literal 2.4.2(1)
  in Interfaces.COBOL B.4(10)
binary adding operator 4.5.3(1)
binary literal 2.4.2(1)
binary operator 4.5(9)
binary_adding_operator 4.5(4)
  used 4.4(4), P.1
Binary_Format
  in Interfaces.COBOL B.4(24)
bit field
  See record_representation_clause 13.5.1(1)
bit ordering 13.5.3(2)
bit string
  See logical operators on boolean arrays 4.5.1(2)
Bit_Order
  in System 13.7(15/2)
Bit_Order aspect 13.5.3(4)
Bit_Order attribute 13.5.3(4)
Bit_Order clause 13.3(7/2), 13.5.3(4)
blank
  in text input for enumeration and numeric types A.10.6(5/2)
Blank_When_Zero
  in Ada.Text_IO.Editing F.3.3(7)
block_statement 5.6(2)
  used 5.1(5/5), P.1
blocked
  [*partial*] D.2.1(11)
  a task state 9(10/5)
  during an entry call 9.5.3(19)
  execution of a selective_accept 9.7.1(16)
  on a delay_statement 9.6(21)
  on an accept_statement 9.5.2(24/5)
  waiting for activations to complete 9.2(5)
  waiting for dependents to terminate 9.3(5)
blocked interrupt C.3(2)
blocking
  allows 9.5(21/5)
  non- 9.5(21/5)
blocking, potentially 9.5(34/5)
  delay_statement 9.6(34), D.9(5)
  remote subprogram call E.4(17/5)
BMP 3.5.2(2/6), 3.5.2(3/6)
BNF (Backus-Naur Form)
  complete listing P.1(1/5)
  cross reference P.2(1/3)
  notation 1.1.4(3)
  under Syntax heading 1.1.2(25)
body 3.11(5), 3.11.1(1/3)
  used 3.11(3), P.1
body_stub 10.1.3(2)
  used 3.11(5), P.1
Body_Version attribute E.3(4)
BOM_16
  in Ada.Strings.UTF_Encoding A.4.11(12/3)
BOM_16BE
  in Ada.Strings.UTF_Encoding A.4.11(10/3)
BOM_16LE
  in Ada.Strings.UTF_Encoding A.4.11(11/3)
BOM_8
  in Ada.Strings.UTF_Encoding A.4.11(9/3)
Boolean 3.5.3(1)
  in Standard A.1(5)
boolean type 3.5.3(1)
boundary entity 7.3.2(8.3/5)
Bounded
  child of Ada.Streams.Storage 13.13.1(26/5)
  child of Ada.Strings A.4.4(3/5)
  child of Ada.Strings.Text_Buffers A.4.12(26/5)
bounded error 1.1.2(31), 1.1.5(8/5)

cause 4.2.1(17/5), 4.5.10(34/6), 4.8(11.1/2), 5.1(18/5), 5.5.3(28/5), 6.2(12/3), 7.6.1(14/1), 9.4(20.1/2), 9.5.1(8/5), 9.5.1(18/5), 9.8(20/5), 10.2(26), 13.9.1(9), 13.11.2(11), 13.11.2(15.2/4), A.5.2(40.1/5), A.17(25/2), A.18.2(238/3), A.18.2(239.1/3), A.18.2(239.2/3), A.18.2(239/2), A.18.2(243/2), A.18.3(152.1/3), A.18.3(152.2/3), A.18.3(152/2), A.18.4(75.7/3), A.18.4(75.8/3), A.18.7(96.19/3), A.18.7(96.20/3), A.18.10(220/3), A.18.10(221/3), A.18.18(68/3), A.18.18(69/3), A.18.19(10/3), A.18.20(14/3), A.18.21(15/3), A.18.22(12/3), A.18.23(15/3), A.18.24(12/3), A.18.25(14/3), A.18.32(11/5), C.7.1(17/3), C.7.2(13.2/1), D.3(13.6/5), E.1(10/2), E.3(6), J.7.1(11)
Bounded_Doubly_Linked_Lists
child of Ada.Containers A.18.20(1/3)
Bounded_Hashed_Maps
child of Ada.Containers A.18.21(1/3)
Bounded_Hashed_Sets
child of Ada.Containers A.18.23(1/3)
Bounded_Indefinite_Holders
child of Ada.Containers A.18.32(1/5)
Bounded_IO
child of Ada.Text_IO A.10.11(3/5)
Bounded_Multiway_Trees
child of Ada.Containers A.18.25(1/3)
Bounded_Ordered_Maps
child of Ada.Containers A.18.22(1/3)
Bounded_Ordered_Sets
child of Ada.Containers A.18.24(1/3)
Bounded_Priority_Queues
child of Ada.Containers A.18.31(2/6)
Bounded_Slice
in Ada.Strings.Bounded A.4.4(28.1/2), A.4.4(28.2/2)
Bounded_String
in Ada.Strings.Bounded A.4.4(6)
Bounded_Synchronized_Queues
child of Ada.Containers A.18.29(2/6)
Bounded_Vectors
child of Ada.Containers A.18.19(1/3)
bounds
of a discrete_range 3.6.1(6)
of an array 3.6(13)
of the index range of an array_aggregate 4.3.3(24)
box
compound delimiter 3.6(15/5)
BPH
in Ada.Characters.Latin_1 A.3.3(17)
broadcast signal
See protected object 9.4(1)
See requeue 9.5.4(1)
Broken_Bar
in Ada.Characters.Latin_1 A.3.3(21/3)
BS
in Ada.Characters.Latin_1 A.3.3(5)
budget D.14.2(14/2)
Budget_Has_Expired
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Budget_Remaining
in Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Buffer_Size
in Ada.Storage_IO A.9(4)
Buffer_Type
in Ada.Strings.Text_Buffers.Bounded A.4.12(27/5)
in Ada.Strings.Text_Buffers.Unbounded A.4.12(19/5)
Buffer_Type *subtype of* Storage_Array
in Ada.Storage_IO A.9(4)
build-in-place
See built in place
built in place 7.6(17.1/5)
by copy parameter passing 6.2(2)
by reference parameter passing 6.2(2)
by-copy type 6.2(3/3)
by-reference primitive 13.1(10/5)
by-reference type 6.2(4)
atomic or volatile C.6(18)
Byte
in Interfaces.COBOL B.4(29/3)
See storage element 13.3(8)
byte sex
See ordering of storage elements in a word 13.5.3(5)
Byte_Array
in Interfaces.COBOL B.4(29/3)

C

C
calling convention B.3(61/3)
child of Interfaces B.3(4/5)
C interface B.3(1/4)
C standard 1.2.1(8/5)
C++ standard 1.2.1(9/5)
C_bool
in Interfaces.C B.3(13.1/5)
C_float
in Interfaces.C B.3(15)
C_Pass_By_Copy
calling convention B.3(60.14/2)
C_Variadic B.3(60.16/4)
calculated
upper bound 3.6.1(9/6)
Calendar
child of Ada 9.6(10/5)
Calendar_Assertion_Check 11.5(23.4/5)
call 6(2/3)
master of 3.10.2(10.1/3)
call on a dispatching operation 3.9.2(2/3)
callable 9.9(2)
Callable attribute 9.9(2)
callable construct 6(2/3)
callable entity 6(2/3)
callable from preelaborable initialization] 10.2.1(11.11/6)
called partition E.4(1/5)
Caller attribute C.7.1(14/3)
calling convention 6.3.1(2/1), B.1(11/3)
Ada 6.3.1(3/3)
associated with a designated profile 3.10(11)
C B.3(61/3)
C_Pass_By_Copy B.3(60.14/2)
C_Variandic_n B.3(60.16/4)
COBOL B.4(1/3)
entry 6.3.1(13/4)
Fortran B.5(1/3)
Intrinsic 6.3.1(4)
protected 6.3.1(12/4)
calling partition E.4(1/5)
calling stub E.4(10)
CAN
in Ada.Characters.Latin_1 A.3.3(6)
can cause 13.1(28/5)
cancel
a logical thread 5.1(16/5)
a parallel construct 5.1(16/5)
Cancel_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Real_Time.Timing_Events D.15(5/2)
cancellation
of a delay_statement 9.6(22/3)
of an entry call 9.5.3(20)
cancellation of a remote subprogram call E.4(13)
canonical form A.5.3(3)
canonical order of array components 5.5.2(11/6)
canonical semantics 11.6(2/3)
canonical-form representation A.5.3(10)
capacity
of a hashed map A.18.5(41/2)
of a hashed set A.18.8(63/2)
of a queue A.18.27(10/3)
of a vector A.18.2(2/2)
in Ada.Containers.Hashed_Maps A.18.5(8/5)
in Ada.Containers.Hashed_Sets A.18.8(10/5)

in Ada.Containers.Vectors A.18.2(19/5)
Capacity_Error
in Ada.Containers A.18.1(5.1/3)
case expression 4.5.7(1/3)
case insensitive 2.3(5/6)
case_expression 4.5.7(5/3)
used 4.5.7(2/3), P.1
case_expression_alternative 4.5.7(6/3)
used 4.5.7(5/3), P.1
case_statement 5.4(2/3)
used 5.1(5/5), P.1
case_statement_alternative 5.4(3)
used 5.4(2/3), P.1
cast
See type conversion 4.6(1/3)
See unchecked type conversion 13.9(1)
catch (an exception)
See handle 11(1/3)
categorization aspect E.2(2/3)
categorization pragma J.15.15(8/5)
Pure J.15.15(9/5)
Remote_Call_Interface J.15.15(9/5)
Remote_Types J.15.15(9/5)
Shared_Passive J.15.15(9/5)
categorized library unit E.2(2/3)
category
of types 3.2(2/5), 3.4(1.1/2)
category determined for a formal type 12.5(6/3)
category of types 1.3.1(13/5)
catenation operator
See concatenation operator 4.5(1)
See concatenation operator 4.5.3(3)
Cause_Of_Termination
in Ada.Task_Termination C.7.3(3/2)
CCH
in Ada.Characters.Latin_1 A.3.3(18)
cease to exist
object 7.6.1(11/6), 13.11.2(10/4)
type 7.6.1(11/6)
Cedilla
in Ada.Characters.Latin_1 A.3.3(22)
Ceiling
in Ada.Containers.Ordered_Maps A.18.6(41/5)
in Ada.Containers.Ordered_Sets A.18.9(51/2), A.18.9(71/5)
Ceiling attribute A.5.3(33), A.5.4(7/6)
ceiling priority
of a protected object D.3(8/3)
Ceiling_Check
[*partial*] C.3.1(11/3), D.3(13), D.3(13.5/5)
Ceiling_Locking locking policy D.3(7)
Cent_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
change of representation 13.6(1/5)
char
in Interfaces.C B.3(19)
char16_array
in Interfaces.C B.3(39.5/3)
char16_nul
in Interfaces.C B.3(39.3/2)
char16_t
in Interfaces.C B.3(39.2/2)
char32_array
in Interfaces.C B.3(39.14/3)
char32_nul
in Interfaces.C B.3(39.12/2)
char32_t
in Interfaces.C B.3(39.11/2)
char_array
in Interfaces.C B.3(23/3)
char_array_access
in Interfaces.C.Strings B.3.1(4)
CHAR_BIT
in Interfaces.C B.3(6)
Character 3.5.2(2/6)
used 2.7(2), P.1
in Standard A.1(35/3)
character encoding A.4.11(46/3)
character plane 2.1(1/6)
character set 2.1(1/6)
character set standard
16 and 32-bit 1.2(8/6), 1.2(9/6)
7-bit 1.2.1(2/5)
8-bit 1.2.1(7/5)
control functions 1.2.1(5/5)
character type 1.3.1(15/5), 3.5.2(1)
character_literal 2.5(2)
used 3.5.1(4), 4.1(2/5), 4.1.3(3), P.1
Character_Mapping
in Ada.Strings.Maps A.4.2(20/5)
Character_Mapping_Function
in Ada.Strings.Maps A.4.2(25)
Character_Range
in Ada.Strings.Maps A.4.2(6)
Character_Ranges
in Ada.Strings.Maps A.4.2(7)
Character_Sequence *subtype of* String
in Ada.Strings.Maps A.4.2(16)
Character_Set
in Ada.Strings.Maps A.4.2(4/5)
in Ada.Strings.Wide_Maps A.4.7(46/2)
in Ada.Strings.Wide_Maps.Wide_-Constants A.4.8(48/2)
in Interfaces.Fortran B.5(11)
Character_Set_Version
in Ada.Wide_Characters.Handling A.3.5(4/3)
characteristics
[*partial*] 3.4(7/3)
Characters
child of Ada A.3.1(2/5)
Characters_Assertion_Check 11.5(23.5/5)
chars_ptr
in Interfaces.C.Strings B.3.1(5/5)
chars_ptr_array
in Interfaces.C.Strings B.3.1(6/2)
check 1.3.5(1/5)
language-defined 11.5(2/3), 11.6(1/3)
check, language-defined
Access_Check 4.1(13), 4.1.5(8/3), 4.6(51/5), 4.8(10.4/3)
Accessibility_Check 3.10.2(29), 4.6(39.1/2), 4.6(48/3), 4.8(10.1/6), 6.5(8/5), 6.5(21/3), 9.5.4(7.2/5), 13.11.4(25/3), 13.11.4(26/3), E.4(18/1)
Allocation_Check 4.8(10.2/2), 4.8(10.3/2), 4.8(10.4/3), 13.11.4(30/3)
Ceiling_Check C.3.1(11/3), D.3(13), D.3(13.5/5)
controlled by assertion policy 3.2.4(31/5), 4.6(51/5), 6.1.1(32/3), 6.1.1(33/3), 6.1.1(35/3), 7.3.2(22/3), 7.3.3(9/5)
Discriminant_Check 4.1.3(15), 4.3(6/6), 4.3.2(8/3), 4.3.4(16/5), 4.6(43), 4.6(45), 4.6(51/5), 4.6(52), 4.7(4/5), 4.8(10/2), 6.5(5.12/5)
Division_Check 3.5.4(20), 4.5.5(22), A.5.3(47), G.1.1(40), K.2(202)
Elaboration_Check 3.11(9)
Index_Check 4.1.1(7), 4.1.2(7), 4.3.3(20.5/6), 4.3.3(29/3), 4.3.3(30), 4.3.4(20/5), 4.5.3(8/6), 4.6(51/5), 4.7(4/5), 4.8(10/2)
Length_Check 4.5.1(8), 4.6(37), 4.6(52)
Overflow_Check 3.5.4(20), 4.4(11), G.2.1(11), G.2.2(7), G.2.3(25), G.2.4(2), G.2.6(3/5)
Partition_Check E.4(19)
Program_Error_Check 3.2.4(29.1/4), 5.5(8.1/5), 5.6.1(4/5), 12.5.1(23.3/2), 13.3(75.1/6), B.3.3(22/2), H.4(23.11/5)
Range_Check 3.2.2(11), 3.5(24), 3.5(27), 3.5(39.12/3), 3.5(39.4/3), 3.5(39.5/3), 3.5(43/5), 3.5(55/5), 3.5.5(7), 3.5.9(19/4), 4.2(11/6), 4.3.3(28), 4.5.1(8), 4.5.6(6), 4.5.6(13), 4.6(28), 4.6(38), 4.6(46), 4.6(51/5), 4.7(4/5), 13.13.2(35/3), A.5.3(26), A.5.3(29), A.5.3(50), A.5.3(53), A.5.3(59), A.5.3(62), K.2(11), K.2(114), K.2(122), K.2(184), K.2(220), K.2(241), K.2(41), K.2(47)
Reserved_Check C.3.1(10/3)
Storage_Check 11.1(6/5), 13.3(67), 13.11(17), D.7(17/1), D.7(18/1), D.7(19/1)
Tag_Check 3.9.2(16), 4.6(42), 4.6(52), 5.2(10), 6.5(8.2/5)
Tasking_Check 9.2(5), 9.5.3(21)
checking pragmas 11.5(1/2)

child
  of a library unit 10.1.1(1)
Child_Count
  in Ada.Containers.Multiway_Trees A.18.10(46.1/5), A.18.10(46/5)
Child_Depth
  in Ada.Containers.Multiway_Trees A.18.10(47.1/5), A.18.10(47/5)
choice parameter 11.2(9)
choice_expression 4.4(2.1/3)
  used 3.8.1(5/3), P.1
choice_parameter_specification 11.2(4)
  used 11.2(3/5), P.1
choice_relation 4.4(2.2/3)
  used 4.4(2.1/3), P.1
chunk 5.5(8.1/5)
  [*partial*] 5.6.1(5/5)
chunk parameter 5.5(6.1/5)
Chunk_Count
  in Ada.Iterator_Interfaces 5.5.1(4.5/5)
Chunk_Index *subtype of* Positive
  in Ada.Iterator_Interfaces 5.5.1(4.2/5)
chunk_specification 5.5(3.1/5)
  used 4.5.10(3/5), 5.5(3/5), 5.6.1(2/5), P.1
Circumflex
  in Ada.Characters.Latin_1 A.3.3(12)
class
  of types 3.2(2/5), 3.4(1.1/2)
  See also package 7(1)
  See also tag 3.9(3)
Class attribute 3.9(14), 7.3.1(9), J.11(2/2)
class factory 3.9(30/2)
class of types 1.3.1(16/5)
class-wide aspect 13.1.1(28/4)
class-wide postcondition expression 6.1.1(5/5)
class-wide precondition expression 6.1.1(3/5)
class-wide stable property function 7.3.4(6/5)
class-wide type 3.4.1(4), 3.7(26)
class-wide type invariant 7.3.2(3/5)
cleanup
  See finalization 7.6.1(1)
clear
  execution timer object D.14.1(12/2)
  group budget object D.14.2(15/2)
  timing event object D.15(9/2)
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(13/5)
  in Ada.Containers.Hashed_Maps A.18.5(12/5)
  in Ada.Containers.Hashed_Sets A.18.8(14/5)
  in Ada.Containers.Indefinite_Holders A.18.18(11/5)
  in Ada.Containers.Multiway_Trees A.18.10(23/5)
  in Ada.Containers.Ordered_Maps A.18.6(11/5)
  in Ada.Containers.Ordered_Sets A.18.9(13/5)
  in Ada.Containers.Vectors A.18.2(24/5)
  in Ada.Environment_Variables A.17(7/2)
  in Ada.Streams.Storage 13.13.1(15/5)
  in Ada.Streams.Storage.Bounded 13.13.1(31/5)
  in Ada.Streams.Storage.Unbounded 13.13.1(23/5)
cleared
  termination handler C.7.3(9/2)
clock 9.6(6/3)
  in Ada.Calendar 9.6(12)
  in Ada.Execution_Time D.14(5/2)
  in Ada.Execution_Time.Interrupts D.14.3(3/5)
  in Ada.Real_Time D.8(6)
clock jump D.8(32)
clock tick D.8(23)
Clock_For_Interrupts
  in Ada.Execution_Time D.14(9.3/3)
Close
  in Ada.Direct_IO A.8.4(8)
  in Ada.Sequential_IO A.8.1(8)
  in Ada.Streams.Stream_IO A.12.1(10)
  in Ada.Text_IO A.10.1(11)
close result set G.2.3(5)
closed entry 9.5.3(5)
  of a protected object 9.5.3(7/3)
  of a task 9.5.3(6/3)
closed under derivation 1.3.1(16/5), 3.4(28)
closure
  downward 3.10.2(37/5)
COBOL
  calling convention B.4(1/3)
  child of Interfaces B.4(7/5)
COBOL interface B.4(1/3)
COBOL standard 1.2.1(4/5)
COBOL_Character
  in Interfaces.COBOL B.4(13)
COBOL_To_Ada
  in Interfaces.COBOL B.4(15)
code point
  for characters 3.5.2(2/6)
code_statement 13.8(2)
  used 5.1(4/2), P.1
Coding aspect 13.4(7)
coextension
  of an object 3.10.2(14.4/3)
Col
  in Ada.Text_IO A.10.1(37/5)
collection
  of an access type 7.6.1(11.1/3)
colon 2.1(15/6)
  in Ada.Characters.Latin_1 A.3.3(10)
column number A.10(9)
comma 2.1(15/6)
  in Ada.Characters.Latin_1 A.3.3(8)
Command_Line
  child of Ada A.15(3/5)
Command_Name
  in Ada.Command_Line A.15(6)
comment 2.7(2)
comments, instructions for submission 0.2(57.22/5)
Commercial_At
  in Ada.Characters.Latin_1 A.3.3(10)
Communication_Error
  in System.RPC E.5(5)
comparison operator
  See relational operator 4.5.2(1)
compatibility
  composite_constraint with an access subtype 3.10(15/2)
  constraint with a subtype 3.2.2(12)
  delta_constraint with an ordinary fixed point subtype J.3(9/4)
  digits_constraint with a decimal fixed point subtype 3.5.9(18/4)
  digits_constraint with a floating point subtype J.3(10/4)
  discriminant constraint with a subtype 3.7.1(10)
  index constraint with a subtype 3.6.1(7)
  range with a scalar subtype 3.5(8)
  range_constraint with a scalar subtype 3.5(8)
compatible
  a type, with a convention B.1(12)
compilation 10.1.1(2)
  separate 10.1(1)
compilation unit 1.3.3(2/5), 10.1(2), 10.1.1(9)
compilation units needed
  by a compilation unit 10.2(2/5)
  remote call interface E.2.3(18)
  shared passive library unit E.2.1(11)
compilation_unit 10.1.1(3)
  used 10.1.1(2), P.1
compile-time error 1.1.2(27), 1.1.5(4)
compile-time semantics 1.1.2(28)
complete context 8.6(4)
completely defined 3.11.1(8)
completion
  abnormal 7.6.1(2/2)
  compile-time concept 3.11.1(1/3)
  normal 7.6.1(2/2)
  run-time concept 7.6.1(2/2)
completion and leaving (completed and left) 7.6.1(2/2)
completion legality
  [*partial*] 3.10.1(13/5)
  entry_body 9.5.2(16)

Complex
  in Ada.Numerics.Generic_Complex_-Types G.1.1(3)
  in Interfaces.Fortran B.5(9)
Complex_Arrays
  child of Ada.Numerics G.3.2(53/2)
Complex_Elementary_Functions
  child of Ada.Numerics G.1.2(9/1)
Complex_IO
  child of Ada.Text_IO G.1.3(3/5)
  child of Ada.Wide_Text_IO G.1.4(1)
  child of Ada.Wide_Wide_Text_IO G.1.5(1/2)
Complex_Matrix
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(4/2)
Complex_Text_IO
  child of Ada G.1.3(9.1/2)
Complex_Types
  child of Ada.Numerics G.1.1(25/1)
Complex_Vector
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(4/2)
component 3.2(2/5)
  of a type 3.2(6/2)
  of the nominal subtype 3.9.1(4.2/5)
component subtype 3.6(10)
component_choice_list 4.3.1(5/5)
  used 4.3.1(4/2), P.1
component_clause 13.5.1(3)
  used 13.5.1(2/5), P.1
component_declaration 3.8(6/3)
  used 3.8(5/1), 9.4(6), P.1
component_definition 3.6(7/2)
  used 3.6(3), 3.6(5), 3.8(6/3), P.1
component_item 3.8(5/1)
  used 3.8(4), P.1
component_list 3.8(4)
  used 3.8(3/5), 3.8.1(3), P.1
Component_Size aspect 13.3(70)
Component_Size attribute 13.3(69)
Component_Size clause 13.3(7/2), 13.3(70)
components
  of a record type 3.8(9/2)
Compose
  in Ada.Directories A.16(20/5)
  in Ada.Directories.Hierarchical_File_Names A.16.1(14/3)
Compose attribute A.5.3(24)
Compose_From_Cartesian
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(9/2), G.3.2(29/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(8)
Compose_From_Polar
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(11/2), G.3.2(32/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(11)
composite type 1.3.1(18/5), 3.2(2/5)
composite_constraint 3.2.2(7)
  used 3.2.2(5), P.1
compound delimiter 2.2(10)
compound_statement 5.1(5/5)
  used 5.1(3), P.1
concatenation operator 4.5(1), 4.5.3(3)
concrete subprogram
  See nonabstract subprogram 3.9.3(1/2)
concrete type
  See nonabstract type 3.9.3(1/2)
concurrent
  actions 9.10(16/5)
concurrent processing
  See parallel construct 5.1(1/5)
  See task 9(1/5)
condition 4.5.7(4/3)
  used 4.5.7(3/3), 5.3(2), 5.5(3/5), 5.5(4.1/5), 5.7(2), 9.5.2(7), 9.7.1(3), P.1
  See also exception 11(1/3)
conditional expression 4.5.7(1/3)
conditional_entry_call 9.7.3(2)
  used 9.7(2), P.1
conditional_expression 4.5.7(2/3)
  used 4.4(7/5), P.1
conditionally evaluated
  subexpression 6.1.1(22.16/5)
conditionally executed 5.5(6.3/5)
conditionally produces
  iteration 5.5(6.4/5)
configuration
  of the partitions of a program E(4)
configuration pragma 10.1.5(8)
  [*partial*] 2.8(6/6)
  Assertion_Policy 11.4.2(7/3)
  Detect_Blocking H.5(4/2)
  Discard_Names C.5(4)
  Locking_Policy D.3(5)
  Normalize_Scalars H.1(4)
  Partition_Elaboration_Policy H.6(5/2)
  Priority_Specific_Dispatching D.2.2(4/2)
  Profile 13.12(14/3)
  Queuing_Policy D.4(5)
  Restrictions 13.12(8/3)
  Reviewable H.3.1(4)
  Suppress 11.5(5/2)
  Task_Dispatching_Policy D.2.2(4/2)
  Unsuppress 11.5(5/2)
  usable as 2.8(6/6)
confirming
  aspect specification 13.1(18.2/5)
  nonoverridable aspect 13.1.1(18.5/6)
  operational aspect 13.1(18.2/5)
  representation aspect 13.1(18.2/5)
  representation item 13.1(18.2/5)
  representation value 13.1(18.2/5)
conflict 9.10(17/5)
  potentially 9.10(17/5)
conflict check policy 9.10.1(1/5)
Conflict_Check_Policy pragma 9.10.1(3/5), L(6.2/5)
conformance 6.3.1(1)
  of an implementation 1.1.3(1)
  See also full conformance, mode conformance, subtype conformance, type conformance
Conjugate
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(13/2), G.3.2(34/2)
  in Ada.Numerics.Generic_Complex_-Types G.1.1(12), G.1.1(15)
consistency
  among compilation units 10.1.4(5)
constant 3.3(13/3)
  result of a function_call 6.4(12/2)
  See also literal 4.2(1)
  See also static 4.9(1)
constant indexing 4.1.6(12/3)
constant object 3.3(13/3)
constant view 3.3(13/3)
Constant_Indexing aspect 4.1.6(2/5)
Constant_Reference
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(17.3/5)
  in Ada.Containers.Hashed_Maps A.18.5(17.3/5), A.18.5(17.5/5)
  in Ada.Containers.Hashed_Sets A.18.8(17.3/5), A.18.8(58.3/5)
  in Ada.Containers.Indefinite_Holders A.18.18(18/5)
  in Ada.Containers.Multiway_Trees A.18.10(30/5)
  in Ada.Containers.Ordered_Maps A.18.6(16.3/5), A.18.6(16.5/5)
  in Ada.Containers.Ordered_Sets A.18.9(16.3/5), A.18.9(73.3/5)
  in Ada.Containers.Vectors A.18.2(34.3/5), A.18.2(34.5/5)
Constant_Reference_Type
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(51.10/6)
  in Ada.Containers.Hashed_Maps A.18.5(37.11/6)
  in Ada.Containers.Hashed_Sets A.18.8(59.10/6)
  in Ada.Containers.Indefinite_Holders A.18.18(16/6)
  in Ada.Containers.Multiway_Trees A.18.10(70.10/6)
  in Ada.Containers.Ordered_Maps A.18.6(51.12/6)
  in Ada.Containers.Ordered_Sets A.18.9(74.10/6)

in Ada.Containers.Vectors A.18.2(79.10/6)
Constants
child of Ada.Strings.Maps A.4.6(3/5)
constituent
of a construct 1.1.4(17)
operative 4.4(9.2/5)
constrained 3.2(9)
known to be 3.3(23.1/3)
object 3.3.1(9/2)
object 6.4.1(16)
subtype 3.2(9), 3.4(6), 3.5(7), 3.5.1(10), 3.5.4(9), 3.5.4(10), 3.5.7(11), 3.5.9(13), 3.5.9(16), 3.6(15/5), 3.6(16), 3.7(26), 3.9(15)
subtype 3.10(14/3)
subtype K.2(33)
Constrained attribute 3.7.2(3/5), J.4(2)
constrained by its initial value 3.3.1(9/2)
[*partial*] 4.8(6/3), 6.5(5.12/5)
constrained_array_definition 3.6(5)
used 3.6(2), P.1
constraint 3.2.2(5)
[*partial*] 3.2(7/2)
null 3.2(7/2)
of a first array subtype 3.6(16)
of a subtype 3.2(8/5)
of an object 3.3.1(9/2)
used 3.2.2(3/2), P.1
Constraint_Error
raised by calculation of upper bound 3.6.1(12/6)
raised by case expression 4.5.7(21/3)
raised by case statement 5.4(13)
raised by detection of a bounded error 13.9.1(9), 13.11.2(15.2/4), A.18.2(238/3), A.18.2(239.2/3), A.18.2(246/2), A.18.3(152.2/3), A.18.4(75.8/3), A.18.7(96.20/3), A.18.10(221/3), A.18.18(69/3)
raised by discriminant-dependent aggregate 4.3.1(19/5)
raised by failure of runtime check 3.2.2(12), 3.5(24), 3.5(27), 3.5(39.12/3), 3.5(39.4/3), 3.5(39.5/3), 3.5(43/5), 3.5(55/5), 3.5.4(20), 3.5.5(7), 3.5.9(19/4), 3.9.2(16), 4.1(13), 4.1.1(7), 4.1.2(7), 4.1.3(15), 4.1.5(8/3), 4.2(11/6), 4.3(6/6), 4.3.2(8/3), 4.3.3(20.5/6), 4.3.3(31), 4.3.4(16/5), 4.3.4(20/5), 4.4(11), 4.5(10), 4.5(11), 4.5(12), 4.5.1(8), 4.5.3(8/6), 4.5.5(22), 4.5.6(6), 4.5.6(13), 4.6(28), 4.6(57/5), 4.6(60), 4.7(4/5), 4.8(10.4/3), 4.8(10/2), 5.2(10), 6.5(5.12/5), 6.5(8.2/5), 11.1(4), 11.5(10/5), 13.13.2(35/3), A.5.3(26), A.5.3(29), A.5.3(47), A.5.3(50), A.5.3(53), A.5.3(59), A.5.3(62), E.4(19), G.1.1(40), G.2.1(12), G.2.2(7), G.2.3(26), G.2.4(3), G.2.6(4), K.2(11), K.2(114), K.2(122), K.2(184), K.2(202), K.2(220), K.2(241), K.2(261), K.2(41), K.2(47)
raised by object of discriminated type 3.8.1(21.1/3)
in Standard A.1(46)
Construct 1.1.4(16), 1.3.3(4/5)
master 7.6.1(3/6)
parallel 5.1(1/5)
constructor
See initialization 3.3.1(18/2)
See initialization 7.6(1)
See initialization expression 3.3.1(4)
See Initialize 7.6(1)
See initialized allocator 4.8(4)
constructor function 3.2.3(8/6)
Constructor_Extension aspect 3.9.1(4.4/6)
container 1.3.3(5/5)
cursor A.18(2/5)
list A.18.3(1/2)
map A.18.4(1/2)
set A.18.7(1/2)
vector A.18.2(1/2)
container aggregate 1.3.3(7/5), 4.3.5(1/5)
container element iterator 5.5.2(3/3)
container type
aggregate aspect 4.3.5(5/5)
container_aggregate 4.3.5(13/5)
used 4.3(2/5), P.1
container_element_association 4.3.5(18/5)
used 4.3.5(17/5), P.1
container_element_association_list 4.3.5(17/5)
used 4.3.5(16/5), P.1
Containers
child of Ada A.18.1(3/5)
Containers_Assertion_Check 11.5(23.6/5)
Containing_Directory
in Ada.Directories A.16(17/5)
in Ada.Directories.Hierarchical_File_Names A.16.1(11/3)
Contains
in Ada.Containers.Doubly_Linked_Lists A.18.3(43/2)
in Ada.Containers.Hashed_Maps A.18.5(32/2)
in Ada.Containers.Hashed_Sets A.18.8(44/2), A.18.8(57/2)
in Ada.Containers.Multiway_Trees A.18.10(41/3)
in Ada.Containers.Ordered_Maps A.18.6(42/2)
in Ada.Containers.Ordered_Sets A.18.9(52/2), A.18.9(72/2)
in Ada.Containers.Vectors A.18.2(71/2)
context
iterative 4.4(9.8/6)
context free grammar
complete listing P.1(1/5)
cross reference P.2(1/3)
notation 1.1.4(3)
under Syntax heading 1.1.2(25)
context pragma
[*partial*] 2.8(7/6)
Elaborate 10.2.1(23/6)
Elaborate_All 10.2.1(23/6)
context_clause 10.1.2(2)
used 10.1.1(3), P.1
context_item 10.1.2(3)
used 10.1.2(2), P.1
contiguous representation
[*partial*] 13.5.2(5), 13.7.1(12), 13.9(9), 13.9(17/3), 13.11(21.6/5)
Continue
in Ada.Asynchronous_Task_Control D.11(3/5)
control character
a category of Character A.3.2(22)
a category of Character A.3.3(4), A.3.3(15)
See also format_effector 2.1(13/3)
control pragma
[*partial*] 2.8(7.3/6)
Control_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Controlled
in Ada.Finalization 7.6(5/5)
controlled type 1.3.1(19/5), 7.6(2), 7.6(9/2)
controlling access result 3.9.2(2/3)
controlling formal parameter 3.9.2(2/3)
controlling operand 3.9.2(2/3)
controlling result 3.9.2(2/3)

controlling tag
  for a call on a dispatching operation 3.9.2(1/5)
controlling tag value 3.9.2(14)
  for the expression in an assignment_statement 5.2(9)
controlling type
  of a formal_abstract_subprogram_declaration 12.6(8.5/5)
convention 6.3.1(2/1), B.1(11/3)
  See also calling convention
Convention aspect B.1(2/3)
Convention pragma J.15.5(4/3), L(8.1/3)
conversion 4.6(1/3), 4.6(28)
  access 4.6(24.11/2), 4.6(24.22/6), 4.6(24.23/6), 4.6(47)
  arbitrary order 1.1.4(18)
  array 4.6(24.2/2), 4.6(36)
  composite (non-array) 4.6(21/3), 4.6(40)
  enumeration 4.6(21.1/2), 4.6(34)
  numeric 4.6(24.1/2), 4.6(29)
  unchecked 13.9(1)
  value 4.6(5/5)
  view 4.6(5/5)
Conversion_Error
  in Interfaces.COBOL B.4(30)
Conversions
  child of Ada.Characters A.3.4(2/5)
  child of Ada.Strings.UTF_Encoding A.4.11(15/5)
Convert
  in Ada.Strings.UTF_Encoding.Conversions A.4.11(16/3), A.4.11(17/3), A.4.11(18/3), A.4.11(19/3), A.4.11(20/3)
convertible 4.6(4/3)
  required 4.6(24.13/2), 4.6(24.4/2), 8.6(27.1/4)
Copy
  in Ada.Containers.Bounded_Doubly_Linked_Lists A.18.20(10/5)
  in Ada.Containers.Bounded_Hashed_Maps A.18.21(13/5)
  in Ada.Containers.Bounded_Hashed_Sets A.18.23(13/5)
  in Ada.Containers.Bounded_Ordered_Maps A.18.22(10/5)
  in Ada.Containers.Bounded_Ordered_Sets A.18.24(10/5)
  in Ada.Containers.Doubly_Linked_Lists A.18.3(17.6/5), A.18.3(51.9/5)
  in Ada.Containers.Hashed_Maps A.18.5(17.8/5), A.18.5(37.10/5)
  in Ada.Containers.Hashed_Sets A.18.8(17.5/5), A.18.8(59.9/5)
  in Ada.Containers.Indefinite_Holders A.18.18(21/5)
  in Ada.Containers.Multiway_Trees A.18.10(33/5), A.18.10(70.9/5)
  in Ada.Containers.Ordered_Maps A.18.6(16.8/5), A.18.6(51.11/5)
  in Ada.Containers.Ordered_Sets A.18.9(16.5/5), A.18.9(74.9/5)
  in Ada.Containers.Vectors A.18.2(34.8/5), A.18.2(79.9/5)
copy back of parameters 6.4.1(17)
copy parameter passing 6.2(2)
Copy_Array
  in Interfaces.C.Pointers B.3.2(15)
Copy_File
  in Ada.Directories A.16(13/2)
Copy_Local_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(54.1/5)
Copy_Sign attribute A.5.3(51)
Copy_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(54.2/5), A.18.10(54/5)
Copy_Terminated_Array
  in Interfaces.C.Pointers B.3.2(14)
Copyright_Sign
  in Ada.Characters.Latin_1 A.3.3(21/3)
core language 1.1.2(2), 1.3.3(8/5)
corresponding constraint 3.4(6)
corresponding discriminants 3.7(18)
corresponding expression
  class-wide type invariant 7.3.2(8/5)
corresponding expresssion
  class-wide postcondition 6.1.1(18/5)
  class-wide precondition 6.1.1(18/5)
corresponding index
  for an array_aggregate 4.3.3(8)
corresponding subtype 3.4(18/3)
corresponding value
  of the target type of a conversion 4.6(28)
Cos
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Cosh
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Cot
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
Coth
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
Count
  in Ada.Direct_IO A.8.4(4)
  in Ada.Streams.Stream_IO A.12.1(7)
  in Ada.Strings.Bounded A.4.4(48), A.4.4(49/6), A.4.4(50)
  in Ada.Strings.Fixed A.4.3(13), A.4.3(14/6), A.4.3(15)
  in Ada.Strings.Unbounded A.4.5(43), A.4.5(44/6), A.4.5(45)
  in Ada.Text_IO A.10.1(5)
Count attribute 9.9(5)
Count_Type
  in Ada.Containers A.18.1(5/2)
Country
  in Ada.Locales A.19(6/3)
Country code standard 1.2(4.1/5)
Country_Code
  in Ada.Locales A.19(4/4)
Country_Unknown
  in Ada.Locales A.19(5/3)
cover
  a type 3.4.1(9/5)
  of a choice and an exception 11.2(6)
cover a value
  by a discrete_choice 3.8.1(9)
  by a discrete_choice_list 3.8.1(13)
covered
  global aspect 6.1.2(38/5)
CPU aspect D.16(8/3)
CPU clock tick D.14(15/2)
CPU pragma J.15.9(2/3), L(8.2/3)
CPU *subtype of* CPU_Range
  in System.Multiprocessors D.16(4/3)
CPU time
  of a task D.14(11/3)
CPU_Range
  in System.Multiprocessors D.16(4/3)
CPU_Set
  in System.Multiprocessors.Dispatching_Domains D.16.1(9.1/4)
CPU_Tick
  in Ada.Execution_Time D.14(4/2)
CPU_Time
  in Ada.Execution_Time D.14(4/2)
CPU_Time_First
  in Ada.Execution_Time D.14(4/2)
CPU_Time_Last
  in Ada.Execution_Time D.14(4/2)
CPU_Time_Unit
  in Ada.Execution_Time D.14(4/2)
CR
  in Ada.Characters.Latin_1 A.3.3(5)
create 3.1(12)

in Ada.Direct_IO A.8.4(6), A.8.4(18.10/5), A.8.4(18.3/5)
in Ada.Sequential_IO A.8.1(6), A.8.1(15.10/5), A.8.1(15.3/5)
in Ada.Streams.Stream_IO A.12.1(8), A.12.1(26.10/5), A.12.1(26.3/5)
in Ada.Text_IO A.10.1(9), A.10.1(85.10/5), A.10.1(85.3/5)
in System.Multiprocessors.Dispatching_Domains D.16.1(7/4), D.16.1(9.2/4)
Create_Directory
in Ada.Directories A.16(7/2)
Create_Path
in Ada.Directories A.16(9/2)
Create_Subpool
in System.Storage_Pools.Subpools 13.11.4(7/3)
creation
of a protected object C.3.1(10/3)
of a return object 6.5(5.12/5)
of a tag 13.14(20/2)
of a task object D.1(17/4)
of an object 3.3(1)
critical section
See intertask communication 9.5(1)
CSI
in Ada.Characters.Latin_1 A.3.3(19)
Currency_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
current column number A.10(9)
current index
of an open direct file A.8(4)
of an open stream file A.12.1(1.1/1)
current instance
of a generic unit 8.6(18)
of a single protected object 8.6(17.6/6)
of a single task 8.6(17.6/6)
of a subtype 8.6(17/6)
of a type 8.6(17/6)
current line number A.10(9)
current mode
of an open file A.7(7)
current page number A.10(9)
Current size
of a stream file A.12.1(1.1/1)
of an external file A.8(3)
Current_Directory
in Ada.Directories A.16(5/2)
Current_Error
in Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Interrupts C.3.2(6)
in Ada.Real_Time.Timing_Events D.15(5/2)
Current_Indent
in Ada.Strings.Text_Buffers A.4.12(14/5)
Current_Input
in Ada.Text_IO A.10.1(17), A.10.1(20)
Current_Output
in Ada.Text_IO A.10.1(17), A.10.1(20)
Current_State
in Ada.Synchronous_Task_Control D.10(4/5)
Current_Task
in Ada.Task_Identification C.7.1(3/5)
Current_Task_Fallback_Handler
in Ada.Task_Termination C.7.3(5/2)
Current_Use
in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
cursor
ambiguous A.18.2(240/2)
for a container A.18(2/5)
invalid A.18.2(248/2), A.18.3(153/2), A.18.4(76/2), A.18.7(97/2), A.18.10(222/3)
in Ada.Containers.Doubly_Linked_Lists A.18.3(7/5), A.18.3(51.3/5)
in Ada.Containers.Hashed_Maps A.18.5(4/5), A.18.5(37.4/5)
in Ada.Containers.Hashed_Sets A.18.8(4/5), A.18.8(59.3/5)
in Ada.Containers.Multiway_Trees A.18.10(9/5), A.18.10(70.3/5)
in Ada.Containers.Ordered_Maps A.18.6(5/5), A.18.6(51.5/5)
in Ada.Containers.Ordered_Sets A.18.9(5/5), A.18.9(74.3/5)
in Ada.Containers.Vectors A.18.2(9/5), A.18.2(79.3/5)

D

dangling reference 13.11.2(15.1/4)
dangling references
prevention via accessibility rules 3.10.2(3/5)
Data_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Storage_IO A.9(9)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
date and time formatting standard 1.2.1(6/5)
Day
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(23/2)
Day_Count
in Ada.Calendar.Arithmetic 9.6.1(10/2)
Day_Duration *subtype of* Duration
in Ada.Calendar 9.6(11/2)
Day_Name
in Ada.Calendar.Formatting 9.6.1(17/2)
Day_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)
Day_Of_Week
in Ada.Calendar.Formatting 9.6.1(18.1/6), 9.6.1(18/2)
DC1
in Ada.Characters.Latin_1 A.3.3(6)
DC2
in Ada.Characters.Latin_1 A.3.3(6)
DC3
in Ada.Characters.Latin_1 A.3.3(6)
DC4
in Ada.Characters.Latin_1 A.3.3(6)
DCS
in Ada.Characters.Latin_1 A.3.3(18)
deadline
active D.2.6(14/5)
base D.2.6(14/5)
relative D.3(13.2/5)
Deadline *subtype of* Time
in Ada.Dispatching.EDF D.2.6(9/5)
Deallocate
in System.Storage_Pools 13.11(8)
in System.Storage_Pools.Subpools 13.11.4(15/3)
Deallocate_Subpool
in System.Storage_Pools.Subpools 13.11.4(12/3)
deallocation of storage 13.11.2(1)
Decimal
child of Ada F.2(2/5)
decimal digit
a category of Character A.3.2(28)
decimal fixed point type 3.5.9(1), 3.5.9(6)
Decimal_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(14.1/6)
in Interfaces.COBOL B.4(31)
Decimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)

Decimal_Element
in Interfaces.COBOL B.4(12/3)
decimal_fixed_point_definition 3.5.9(4)
used 3.5.9(2), P.1
Decimal_IO
in Ada.Text_IO A.10.1(73)
decimal_literal 2.4.1(2)
used 2.4(2), P.1
Decimal_Output
in Ada.Text_IO.Editing F.3.3(11)
declaration 1.3.3(10/5), 3.1(5), 3.1(6/5)
declaration list
[*partial*] 12.1(9.1/5)
declarative_part 3.11(6.1/2)
package_specification 7.1(6/2)
declarative region
of a construct 8.1(1/6)
declarative_item 3.11(3)
used 3.11(2), P.1
declarative_part 3.11(2)
used 5.6(2), 6.3(2/3), 7.2(2/3), 9.1(6/3), 9.5.2(5/5), P.1
declare 3.1(8), 3.1(12)
declare expression 4.5.9(1/5)
declare_expression 4.5.9(2/5)
used 4.4(7/5), P.1
declare_item 4.5.9(3/5)
used 4.5.9(2/5), P.1
declared pure 10.2.1(17/5)
Decode
in Ada.Strings.UTF_Encoding.Strings A.4.11(26/3), A.4.11(27/3), A.4.11(28.1/6), A.4.11(28.2/6), A.4.11(28.3/6), A.4.11(28/3)
in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(34/3), A.4.11(35/3), A.4.11(36.1/6), A.4.11(36.2/6), A.4.11(36.3/6), A.4.11(36/3)
in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(42/3), A.4.11(43/3), A.4.11(44/3), A.4.11(82.1/6), A.4.11(82.3/6), A.4.11(82.5/6), A.4.11(94.1/6), A.4.11(94.3/6), A.4.11(94.5/6)
Decrease_Indent
in Ada.Strings.Text_Buffers A.4.12(16/5)
Decrement
in Interfaces.C.Pointers B.3.2(11/3)
deeper
accessibility level 3.10.2(3/5)
statically 3.10.2(4), 3.10.2(17)
default constant indexing function 5.5.1(16/3)
default cursor subtype 5.5.1(8/5)
default directory A.16(48/2)
default element subtype 5.5.1(9/5)
default entry queuing policy 9.5.3(17)
default initial condition 1.3.1(21/5)
default initial condition check 7.3.3(9/5)
default initial condition expression 7.3.3(2/5)
default iterator function 5.5.1(8/5)
default iterator subtype 5.5.1(8/5)
default pool 13.11.3(4.2/4)
default treatment C.3(5)
default variable indexing function 5.5.1(21/3)
Default_Aft
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Base
in Ada.Text_IO A.10.1(53), A.10.1(58)
Default_Bit_Order
in System 13.7(15/2)
Default_Component_Value aspect 3.6(22.2/3)
Default_Currency
in Ada.Text_IO.Editing F.3.3(10)
Default_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Default_Exp
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
default_expression 3.7(6)
used 3.7(5/5), 3.8(6/3), 6.1(15/5), 12.4(2/3), P.1
Default_Fill
in Ada.Text_IO.Editing F.3.3(10)
Default_Fore
in Ada.Text_IO A.10.1(64), A.10.1(69), A.10.1(74)
in Ada.Text_IO.Complex_IO G.1.3(5)
Default_Initial_Condition aspect 7.3.3(2/5)
Default_Iterator aspect 5.5.1(8/5)
Default_Modulus
in Ada.Containers.Bounded_Hashed_Maps A.18.21(10/3)
in Ada.Containers.Bounded_Hashed_Sets A.18.23(10/3)
default_name 12.6(4)
used 12.6(3/2), P.1
Default_Priority
in System 13.7(17)
Default_Quantum
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Default_Radix_Mark
in Ada.Text_IO.Editing F.3.3(10)
Default_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Default_Separator
in Ada.Text_IO.Editing F.3.3(10)
Default_Setting
in Ada.Text_IO A.10.1(80)
Default_Storage_Pool aspect 13.11.3(5/4)
Default_Storage_Pool pragma 13.11.3(3/3), L(8.3/3)
Default_Subpool_for_Pool
in System.Storage_Pools.Subpools 13.11.4(13/3)
Default_Value aspect 3.5(56.3/3)
Default_Width
in Ada.Text_IO A.10.1(53), A.10.1(58), A.10.1(80)
deferred constant 7.4(2/3)
deferred constant declaration 3.3.1(6/5), 7.4(2/3)
defining name 3.1(10)
defining_character_literal 3.5.1(4)
used 3.5.1(3), P.1
defining_designator 6.1(6)
used 6.1(4.2/2), 12.3(2/3), P.1
defining_identifier 3.1(4)
used 3.2.1(3/3), 3.2.2(2/3), 3.3.1(3), 3.5.1(3), 3.10.1(2/2), 4.3.3(5.1/5), 5.5(3.1/5), 5.5(4/5), 5.5.2(2/5), 5.5.3(3/5), 6.1(7), 6.5(2.1/5), 7.3(2/3), 7.3(3/3), 8.5.1(2/5), 8.5.2(2/3), 9.1(2/3), 9.1(3/5), 9.1(6/3), 9.4(2/3), 9.4(3/3), 9.4(7/3), 9.5.2(2/3), 9.5.2(5/5), 9.5.2(8/5), 10.1.3(4/3), 10.1.3(5/3), 10.1.3(6/3), 11.2(4), 12.5(2.1/5), 12.5(2.2/5), 12.7(2/3), P.1
defining_identifier_list 3.3.1(3)
used 3.3.1(2/3), 3.3.2(2), 3.7(5/5), 3.8(6/3), 6.1(15/5), 11.1(2/3), 12.4(2/3), P.1
defining_operator_symbol 6.1(11)
used 6.1(6), P.1
defining_program_unit_name 6.1(7)
used 6.1(4.1/2), 6.1(6), 7.1(3/3), 7.2(2/3), 8.5.3(2/3), 8.5.5(2/3), 12.3(2/3), P.1
Definite attribute 12.5.1(23/3)
definite subtype 3.3(23/5)
definition 3.1(7)
Degree_Sign
in Ada.Characters.Latin_1 A.3.3(22)
DEL
in Ada.Characters.Latin_1 A.3.3(14)
delay_alternative 9.7.1(6)
used 9.7.1(4), 9.7.2(2), P.1
delay_relative_statement 9.6(4)
used 9.6(2), P.1
delay_statement 9.6(2)
used 5.1(4/2), 9.7.1(6), 9.7.4(4/2), P.1
Delay_Until_And_Set_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(14/3)

Delay_Until_And_Set_Deadline
  in Ada.Dispatching.EDF D.2.6(9/5)
delay_until_statement 9.6(3)
  used 9.6(2), P.1
Delete
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(24/5)
  in Ada.Containers.Hashed_Maps A.18.5(25/5), A.18.5(26/5)
  in Ada.Containers.Hashed_Sets A.18.8(24/5), A.18.8(25/5), A.18.8(55/5)
  in Ada.Containers.Ordered_Maps A.18.6(24/5), A.18.6(25/5)
  in Ada.Containers.Ordered_Sets A.18.9(23/5), A.18.9(24/5), A.18.9(68/5)
  in Ada.Containers.Vectors A.18.2(50/5), A.18.2(51/5)
  in Ada.Direct_IO A.8.4(8)
  in Ada.Sequential_IO A.8.1(8)
  in Ada.Streams.Stream_IO A.12.1(10)
  in Ada.Strings.Bounded A.4.4(64), A.4.4(65)
  in Ada.Strings.Fixed A.4.3(29), A.4.3(30)
  in Ada.Strings.Unbounded A.4.5(59), A.4.5(60)
  in Ada.Text_IO A.10.1(11)
Delete_Children
  in Ada.Containers.Multiway_Trees A.18.10(53/5)
Delete_Directory
  in Ada.Directories A.16(8/2)
Delete_File
  in Ada.Directories A.16(11/2)
Delete_First
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(25/5)
  in Ada.Containers.Ordered_Maps A.18.6(26/5)
  in Ada.Containers.Ordered_Sets A.18.9(25/5)
  in Ada.Containers.Vectors A.18.2(52/5)
Delete_Last
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(26/5)
  in Ada.Containers.Ordered_Maps A.18.6(27/5)
  in Ada.Containers.Ordered_Sets A.18.9(26/5)
  in Ada.Containers.Vectors A.18.2(53/5)
Delete_Leaf
  in Ada.Containers.Multiway_Trees A.18.10(35/5)
Delete_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(36/5)
Delete_Tree
  in Ada.Directories A.16(10/2)
delimiter 2.2(8/2)
delivery
  of an interrupt C.3(2)
delta
  of a fixed point type 3.5.9(1)
delta aggregate 4.3.4(1/5)
Delta attribute 3.5.10(3)
delta_aggregate 4.3.4(2/5)
  used 4.3(2/5), P.1
delta_constraint J.3(2/4)
  used 3.2.2(6), P.1
Denominator
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(8/5)
Denorm attribute A.5.3(9)
denormalized number A.5.3(10)
denote 8.6(16)
  informal definition 3.1(8)
  name used as a pragma argument 8.6(32)
depend on a discriminant
  for a component 3.7(20)
  for a constraint or component_definition 3.7(19)
dependence
  elaboration 10.2(9/5)
  of a task on a master 9.3(1)
  of a task on another task 9.3(4)
  semantic 10.1.1(26/2)
depth
  accessibility level 3.10.2(3/5)
  in Ada.Containers.Multiway_Trees A.18.10(19.1/5), A.18.10(19/5)
depth-first order A.18.10(5/3)
Dequeue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(6/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
Dequeue_Only_High_Priority
  in Ada.Containers.Bounded_Priority_Queues A.18.31(6/3)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(6/3)
dereference 4.1(8)
Dereference_Error
  in Interfaces.C.Strings B.3.1(12)
derivation class
  for a type 3.4.1(2/2)
derived from
  directly or indirectly 3.4.1(2/2)
derived type 1.3.1(22/5), 3.4(1/2)
  [*partial*] 3.4(24)
derived_type_definition 3.4(2/2)
  used 3.2.1(4/2), P.1
descendant 10.1.1(11)
  at run time 3.9(12.3/3)
  of a tree node A.18.10(4/3)
  of a type 3.4.1(10/2)
  of the full view of a type 7.3.1(5.2/5)
  relationship with scope 8.2(4)
descendant of a type 1.3.1(25/5)
Descendant_Tag
  in Ada.Tags 3.9(7.1/2)
designate 3.10(1)
designated profile
  of an access-to-subprogram type 3.10(11)
  of an anonymous access type 3.10(12/3)
designated subtype
  of a named access type 3.10(10)
  of an anonymous access type 3.10(12/3)
designated type
  of a named access type 3.10(10)
  of an anonymous access type 3.10(12/3)
designator 6.1(5)
  used 6.3(2/3), P.1
destructor
  See finalization 7.6(1)
  See finalization 7.6.1(1)
Detach_Handler
  in Ada.Interrupts C.3.2(9)
Detect_Blocking pragma H.5(3/2), L(8.4/2)
Determinant
  in Ada.Numerics.Generic_Complex_-Arrays G.3.2(46/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
determined category for a formal type 12.5(6/3)
determined to be unevaluated 6.1.1(24.1/5)
determines
  a type by a subtype_mark 3.2.2(8)
determining expression 6.1.1(22.16/5)
Device_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)

in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Diaeresis
in Ada.Characters.Latin_1 A.3.3(21/3)
Difference
in Ada.Calendar.Arithmetic 9.6.1(12/2)
in Ada.Containers.Hashed_Sets A.18.8(32/5), A.18.8(33/5)
in Ada.Containers.Ordered_Sets A.18.9(33/5), A.18.9(34/5)
digit 2.4.1(4.1/2)
used 2.4.1(3), 2.4.2(5), P.1
digits
of a decimal fixed point subtype 3.5.9(6), 3.5.10(7)
Digits attribute 3.5.8(2/1), 3.5.10(7)
digits_constraint 3.5.9(5/4)
used 3.2.2(6), P.1
dimensionality
of an array 3.6(12)
direct access A.8(3)
direct file A.8(1/2)
Direct_IO
child of Ada A.8.4(2/5)
direct_name 4.1(3)
used 3.8.1(2), 4.1(2/5), 5.1(8), 9.5.2(3), 13.1(3), J.7(1), J.15.14(4/5), L(25.3/5), P.1
Direction
in Ada.Strings A.4.1(6)
directly specified
of a representation aspect of an entity 13.1(8/5)
of an operational aspect of an entity 13.1(8.1/3)
directly visible 8.3(2), 8.3(21)
within a pragma in a context_clause 10.1.6(3)
within a pragma that appears at the place of a compilation unit 10.1.6(5)
within a use_clause in a context_clause 10.1.6(3)
within a with_clause 10.1.6(2/2)
within the parent_unit_name of a library unit 10.1.6(2/2)
within the parent_unit_name of a subunit 10.1.6(4)
Directories
child of Ada A.16(3/5)
directory A.16(45/2)
directory entry A.16(49/2)
directory name A.16(46/2)
Directory_Entry_Type
in Ada.Directories A.16(29/2)
disabled
predicate checks 3.2.4(7/3)
Discard_Names aspect C.5(1.3/4)
Discard_Names pragma C.5(3), L(9)
discontiguous representation [*partial*] 13.5.2(5), 13.7.1(12), 13.9(9), 13.9(17/3), 13.11(21.6/5)
discrete array type 4.5.2(1)
discrete type 1.3.1(27/5), 3.2(3), 3.5(1)
discrete_choice 3.8.1(5/3)
used 3.8.1(4/5), P.1
discrete_choice_list 3.8.1(4/5)
used 3.8.1(3), 4.3.3(5.1/5), 4.3.3(5/5), 4.5.7(6/3), 5.4(3), P.1
Discrete_Random
child of Ada.Numerics A.5.2(17/5)
discrete_range 3.6.1(3)
used 3.6.1(2), 4.1.2(2), 4.3.5(20/5), P.1
discrete_subtype_definition 3.6(6)
used 3.6(5), 5.5(3.1/5), 5.5(4/5), 9.5.2(2/3), 9.5.2(8/5), P.1
discriminant 1.3.1(28/5), 3.2(5/2), 3.7(1/2)
of a variant_part 3.8.1(6)
use in a record definition 3.8(12/3)
discriminant_association 3.7.1(3/5)
used 3.7.1(2), P.1
Discriminant_Check 11.5(12)
[*partial*] 4.1.3(15), 4.3(6/6), 4.3.2(8/3), 4.3.4(16/5), 4.6(43), 4.6(45), 4.6(51/5), 4.6(52), 4.7(4/5), 4.8(10/2), 6.5(5.12/5)
discriminant_constraint 3.7.1(2)
used 3.2.2(7), P.1
discriminant_part 3.7(2/2)
used 3.10.1(2/2), 7.3(2/3), 7.3(3/3), 12.5(2.1/5), 12.5(2.2/5), P.1
discriminant_specification 3.7(5/5)
used 3.7(4), P.1
discriminants
known 3.7(26)
unknown 3.7(26)
discriminated type 3.7(8/2)
dispatching 3.9(3)
child of Ada D.2.1(1.2/5)
Dispatching aspect H.7.1(8/5)
dispatching call
on a dispatching operation 3.9.2(1/5)
dispatching domain D.16.1(16/4)
dispatching operation 3.9.2(1/5), 3.9.2(2/3)
[*partial*] 3.9(1)
dispatching operation set H.7.1(14/5)
dispatching point D.2.1(4/5)
[*partial*] D.2.3(8/2), D.2.4(9/3)
dispatching policy for tasks [*partial*] D.2.1(5/2)
dispatching, task D.2.1(4/5)
Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(5/3)
Dispatching_Domain aspect D.16.1(18/3)
Dispatching_Domain pragma J.15.10(2/3), L(9.1/3)
Dispatching_Domain_Error
in System.Multiprocessors.Dispatching_Domains D.16.1(4/3)
Dispatching_Domains
child of System.Multiprocessors D.16.1(3/5)
dispatching_operation_set H.7.1(9/5)
dispatching_operation_specifier H.7.1(10/5)
used H.7.1(9/5), P.1
Dispatching_Policy_Error
in Ada.Dispatching D.2.1(1.4/3)
Display_Format
in Interfaces.COBOL B.4(22)
displayed magnitude (of a decimal value) F.3.2(14)
disruption of an assignment 9.8(21), 13.9.1(5)
[*partial*] 11.6(6/5)
distinct access paths 6.2(12/3)
distributed accessibility 3.10.2(32.1/3)
distributed program E(3)
distributed system E(2)
distributed systems C(1)
divide 2.1(15/6)
in Ada.Decimal F.2(6/3)
divide operator 4.5(1), 4.5.5(1)
Division_Check 11.5(13/2)
[*partial*] 3.5.4(20), 4.5.5(22), A.5.3(47), G.1.1(40), K.2(202)
Division_Sign
in Ada.Characters.Latin_1 A.3.3(26)
DLE
in Ada.Characters.Latin_1 A.3.3(6)
Do_APC
in System.RPC E.5(10)
Do_RPC
in System.RPC E.5(9)
documentation (required of an implementation) 1.1.3(18), M.1(1/5), M.2(1/5), M.3(1/5)
documentation requirements 1.1.2(34), M(1/5)
summary of requirements M.1(1/5)
Dollar_Sign
in Ada.Characters.Latin_1 A.3.3(8)
dot 2.1(15/6)
dot selection
See selected_component 4.1.3(1)
double
in Interfaces.C B.3(16)
Double_Complex
in Interfaces.Fortran B.5(10.2/5)
Double_Imaginary *subtype of* Imaginary
in Interfaces.Fortran B.5(10.3/5)
Double_Precision
in Interfaces.Fortran B.5(6)

Double_Precision_Complex_Types
in Interfaces.Fortran B.5(10.1/5)
Doubly_Linked_Lists
child of Ada.Containers A.18.3(5/5)
downward closure 3.10.2(37/5)
drift rate D.8(41)
Duration
in Standard A.1(43)
dynamic binding
See dispatching operation 3.9(1)
dynamic semantics 1.1.2(30)
Dynamic_Predicate aspect 3.2.4(1/5)
Dynamic_Priorities
child of Ada D.5.1(3/5)
dynamically determined tag 3.9.2(1/5)
dynamically enclosing
of one execution by another 11.4(2)
dynamically tagged 3.9.2(5/2)

E

e
in Ada.Numerics A.5(3/5)
EDF
child of Ada.Dispatching D.2.6(9/5)
child of Ada.Synchronous_Task_Control D.10(5.2/5)
EDF_Within_Priorities task dispatching policy D.2.6(7/5)
edited output F.3(1/2)
Editing
child of Ada.Text_IO F.3.3(3/5)
child of Ada.Wide_Text_IO F.3.4(1)
child of Ada.Wide_Wide_Text_IO F.3.5(1/2)
effect
external 1.1.3(8)
efficiency 11.5(29/5), 11.6(1/3)
Eigensystem
in Ada.Numerics.Generic_Complex_Arrays G.3.2(49/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(27/2)
Eigenvalues
in Ada.Numerics.Generic_Complex_Arrays G.3.2(48/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(26/2)
Elaborate pragma 10.2.1(20), L(10)
Elaborate_All pragma 10.2.1(21), L(11)
Elaborate_Body aspect 10.2.1(26.1/5)
Elaborate_Body pragma J.15.14(6/5), L(12.1/5)
elaborated 3.11(8)
elaboration 1.3.4(3/5), 3.1(11)
abstract_subprogram_declaration 3.9.3(11.1/2)
access_definition 3.10(17/2)
access_type_definition 3.10(16)
array_type_definition 3.6(21)
aspect_clause 13.1(19/1)
choice_parameter_specification 11.4(7)
chunk_specification 5.5(8.1/5)
component_declaration 3.8(17)
component_definition 3.6(22/2), 3.8(18/6)
component_list 3.8(17)
declaration with a True Import aspect B.1(38/3)
declarative_part 3.11(7)
deferred constant declaration 7.4(10/3)
delta_constraint J.3(11)
derived_type_definition 3.4(26)
digits_constraint 3.5.9(19/4)
discrete_subtype_definition 3.6(22/2)
discriminant_constraint 3.7.1(12/5)
entry_declaration 9.5.2(22/1)
enumeration_type_definition 3.5.1(10)
exception_declaration 11.1(5)
expression_function_declaration 6.8(8/3)
fixed_point_definition 3.5.9(17)
floating_point_definition 3.5.7(13)
full type definition 3.2.1(11)
full_type_declaration 3.2.1(11)
generic body 12.2(2)
generic_declaration 12.1(10)
generic_instantiation 12.3(20)
incomplete_type_declaration 3.10.1(12)
index_constraint 3.6.1(8)
integer_type_definition 3.5.4(18)
loop_parameter_specification 5.5(9/5)
nongeneric package_body 7.2(6)
nongeneric subprogram_body 6.3(6)
null_procedure_declaration 6.7(5/3)
number_declaration 3.3.2(7)
object_declaration 3.3.1(15)
of library units for a foreign language main subprogram B.1(39/3)
package_body of Standard A.1(50)
package_declaration 7.1(8)
partition E.1(6)
partition E.5(21)
per-object constraint 3.8(18.1/1)
pragma 2.8(12/6)
private_extension_declaration 7.3(17)
private_type_declaration 7.3(17)
protected declaration 9.4(12)
protected_body 9.4(15)
protected_definition 9.4(13)
range_constraint 3.5(9)
real_type_definition 3.5.6(5)
record_definition 3.8(16)
record_extension_part 3.9.1(5)
record_type_definition 3.8(16)
renaming_declaration 8.5(3)
single_protected_declaration 9.4(12)
single_task_declaration 9.1(10)
subprogram_declaration 6.1(31/2)
subtype_declaration 3.2.2(9)
subtype_indication 3.2.2(9)
task declaration 9.1(10)
task_body 9.1(13)
task_definition 9.1(11)
use_clause 8.4(12)
variant_part 3.8.1(22)
elaboration control 10.2.1(1/5)
elaboration dependence
library_item on another 10.2(9/5)
Elaboration_Check 11.5(20)
[*partial*] 3.11(9)
element
of a storage pool 13.11(11)
in Ada.Containers.Doubly_Linked_Lists A.18.3(14.1/5), A.18.3(14/5)
in Ada.Containers.Hashed_Maps A.18.5(14.1/5), A.18.5(14/5), A.18.5(31/2)
in Ada.Containers.Hashed_Sets A.18.8(15.1/5), A.18.8(15/5), A.18.8(52/2)
in Ada.Containers.Indefinite_Holders A.18.18(12/5)
in Ada.Containers.Multiway_Trees A.18.10(24.1/5), A.18.10(24/5)
in Ada.Containers.Ordered_Maps A.18.6(13.1/5), A.18.6(13/5), A.18.6(39/2)
in Ada.Containers.Ordered_Sets A.18.9(14.1/5), A.18.9(14/5), A.18.9(65/2)
in Ada.Containers.Vectors A.18.2(27/5), A.18.2(28.1/5), A.18.2(28/5)
in Ada.Strings.Bounded A.4.4(26)
in Ada.Strings.Unbounded A.4.5(20)
element type
container aggregate 4.3.5(7/6), 4.3.5(9/6)
Element_Count
in Ada.Streams.Storage 13.13.1(14/5)
in Ada.Streams.Storage.Bounded 13.13.1(30/5)
in Ada.Streams.Storage.Unbounded 13.13.1(22/5)
elementary type 1.3.1(30/5), 3.2(2/5)
Elementary_Functions
child of Ada.Numerics A.5.1(9/1)
eligible
a type, for a convention B.1(14/3)
else part
of a selective_accept 9.7.1(11)
EM
in Ada.Characters.Latin_1 A.3.3(6)
embedded systems C(1), D(1/5)

Empty
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(10.3/5)
  in Ada.Containers.Hashed_Maps A.18.5(7.3/5)
  in Ada.Containers.Hashed_Sets A.18.8(8.2/5)
  in Ada.Containers.Indefinite_Holders A.18.18(8.2/5)
  in Ada.Containers.Multiway_Trees A.18.10(15.3/5)
  in Ada.Containers.Ordered_Maps A.18.6(8.3/5)
  in Ada.Containers.Ordered_Sets A.18.9(9.2/5)
  in Ada.Containers.Vectors A.18.2(12.4/5)
empty element
  of a vector A.18.2(4/2)
empty holder A.18.18(3/3)
Empty_Holder
  in Ada.Containers.Indefinite_Holders A.18.18(7/3)
Empty_List
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(8/2), A.18.3(51.4/5)
Empty_Map
  in Ada.Containers.Hashed_Maps A.18.5(5/2), A.18.5(37.5/5)
  in Ada.Containers.Ordered_Maps A.18.6(6/2), A.18.6(51.6/5)
Empty_Set
  in Ada.Containers.Hashed_Sets A.18.8(5/2), A.18.8(59.4/5)
  in Ada.Containers.Ordered_Sets A.18.9(6/2), A.18.9(74.4/5)
Empty_Tree
  in Ada.Containers.Multiway_Trees A.18.10(10/3), A.18.10(70.4/5)
Empty_Vector
  in Ada.Containers.Vectors A.18.2(10/2), A.18.2(79.4/5)
enabled
  invariant expression 7.3.2(21/4)
  postcondition expression 6.1.1(19/5)
  precondition expression 6.1.1(19/5)
  predicate checks 3.2.4(7/3)
encapsulation
  See package 7(1)
enclosing
  immediately 8.1(13/6)
Encode
  in Ada.Strings.UTF_Encoding.Strings A.4.11(23/3), A.4.11(24/3), A.4.11(25/3)
  in Ada.Strings.UTF_Encoding.Wide_Strings A.4.11(31/3), A.4.11(32/3), A.4.11(33/3)
  in Ada.Strings.UTF_Encoding.Wide_Wide_Strings A.4.11(39/3), A.4.11(40/3), A.4.11(41/3)
Encoding
  in Ada.Strings.UTF_Encoding A.4.11(13/3)
encoding scheme A.4.11(46/3)
Encoding_Error
  in Ada.Strings.UTF_Encoding A.4.11(8/3)
Encoding_Scheme
  in Ada.Strings.UTF_Encoding A.4.11(4/3)
end of a line 2.2(2/3)
End_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
End_Of_File
  in Ada.Direct_IO A.8.4(16)
  in Ada.Sequential_IO A.8.1(13)
  in Ada.Streams.Stream_IO A.12.1(12)
  in Ada.Text_IO A.10.1(34/5)
End_Of_Line
  in Ada.Text_IO A.10.1(30/5)
End_Of_Page
  in Ada.Text_IO A.10.1(33/5)
End_Search
  in Ada.Directories A.16(33/2)
endian
  big 13.5.3(2)
  little 13.5.3(2)
ENQ
  in Ada.Characters.Latin_1 A.3.3(5)
Enqueue
  in Ada.Containers.Bounded_Priority_Queues A.18.31(5/3)
  in Ada.Containers.Bounded_Synchronized_Queues A.18.29(5/3)
  in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(5/5)
  in Ada.Containers.Unbounded_Priority_Queues A.18.30(5/3)
  in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(5/3)
entity
  [*partial*] 3.1(1/5)
entity with runtime name text C.5(1.1/4)
entry
  closed 9.5.3(5)
  open 9.5.3(5)
  single 9.5.2(20)
entry call 9.5.3(1)
  simple 9.5.3(1)
entry calling convention 6.3.1(13/4)
entry family 9.5.2(20)
entry index subtype 3.8(18/6), 9.5.2(20)
entry queue 9.5.3(12)
entry queuing policy 9.5.3(17)
  default policy 9.5.3(17)
entry_barrier 9.5.2(7)
  used 9.5.2(5/5), P.1
entry_body 9.5.2(5/5)
  used 9.4(8/4), P.1
entry_body_formal_part 9.5.2(6)
  used 9.5.2(5/5), P.1
entry_call_alternative 9.7.2(3/2)
  used 9.7.2(2), 9.7.3(2), P.1
entry_call_statement 9.5.3(2)
  used 5.1(4/2), 9.7.2(3.1/2), P.1
entry_declaration 9.5.2(2/3)
  used 9.1(5/1), 9.4(5/1), P.1
entry_index 9.5.2(4)
  used 9.5.2(3), P.1
entry_index_specification 9.5.2(8/5)
  used 9.5.2(6), P.1
Enum_Rep attribute 13.4(10.2/5)
Enum_Val attribute 13.4(10.5/5)
enumeration literal 3.5.1(6/3)
enumeration type 1.3.1(31/5), 3.2(3), 3.5.1(1)
enumeration_aggregate 13.4(3)
  used 13.4(2), P.1
Enumeration_IO
  in Ada.Text_IO A.10.1(79)
enumeration_literal_specification 3.5.1(3)
  used 3.5.1(2), P.1
enumeration_representation_clause 13.4(2)
  used 13.1(2/1), P.1
enumeration_type_definition 3.5.1(2)
  used 3.2.1(4/2), P.1
environment 10.1.4(1)
environment declarative_part 10.1.4(1)
  for the environment task of a partition 10.2(13)
environment task 10.2(8)
environment variable A.17(1/2)
Environment_Task
  in Ada.Task_Identification C.7.1(3/5)
Environment_Variables
  child of Ada A.17(3/5)
EOT
  in Ada.Characters.Latin_1 A.3.3(5)
EPA
  in Ada.Characters.Latin_1 A.3.3(18)
epoch D.8(19)
equal operator 4.5(1), 4.5.2(1)

Equal_Case_Insensitive
child of Ada.Strings A.4.10(2/5)
child of Ada.Strings.Bounded A.4.10(7/5)
child of Ada.Strings.Fixed A.4.10(5/3)
child of Ada.Strings.Unbounded A.4.10(10/5)
Equal_Element
in Ada.Containers.Indefinite_Holders A.18.18(7.1/5)
in Ada.Containers.Multiway_Trees A.18.10(11.1/5)
Equal_Subtree
in Ada.Containers.Multiway_Trees A.18.10(14/3)
equality operator 4.5.2(1)
special inheritance rule for tagged types 3.4(17/2), 4.5.2(14/3)
equals sign 2.1(15/6)
Equals_Sign
in Ada.Characters.Latin_1 A.3.3(10)
equivalent element
of a hashed set A.18.8(64/2)
of an ordered set A.18.9(78/2)
equivalent key
of a hashed map A.18.5(42/2)
of an ordered map A.18.6(55/2)
Equivalent_Elements
in Ada.Containers.Hashed_Sets A.18.8(46/5), A.18.8(47/5), A.18.8(48/5)
in Ada.Containers.Ordered_Sets A.18.9(3/2)
Equivalent_Keys
in Ada.Containers.Hashed_Maps A.18.5(34/5), A.18.5(35/5), A.18.5(36/5)
in Ada.Containers.Ordered_Maps A.18.6(3/5)
in Ada.Containers.Ordered_Sets A.18.9(63/2)
Equivalent_Sets
in Ada.Containers.Hashed_Sets A.18.8(8/2)
in Ada.Containers.Ordered_Sets A.18.9(9/2)
erroneous execution 1.1.2(32), 1.1.5(10/5)
cause 3.7.2(4), 3.9(25.3/2), 6.4.1(18/3), 9.8(21), 9.10(19/5), 11.5(26/5), 13.3(13/3), 13.3(27), 13.3(28/2), 13.9.1(8), 13.9.1(12/3), 13.9.1(13/3), 13.11(21), 13.11.2(16/3), 13.11.4(31.1/4), 13.13.2(53/2), A.10.3(22/1), A.12.1(36.1/1), A.13(17), A.17(28/2), A.18.2(252/2), A.18.3(157/2), A.18.4(80/2), A.18.7(101/2), A.18.18(70/3), A.18.19(11/3), A.18.20(15/3), A.18.21(16/3), A.18.22(13/3), A.18.23(16/3), A.18.24(13/3), A.18.25(15/3), B.1(38.1/5), B.3.1(51), B.3.1(55), B.3.1(56), B.3.1(57), B.3.2(35), B.3.2(36), B.3.2(37), B.3.2(38), B.3.2(39), B.3.2(42), C.3.1(14), C.3.1(14.1/3), C.7.1(18), C.7.2(14), C.7.2(15), C.7.2(15.1/2), D.2.6(31/2), D.5.1(12), D.11(9), D.14(19/2), D.14.1(25/2), D.14.2(35/2), H.4(26), H.4(27)
error
compile-time 1.1.2(27), 1.1.5(4)
link-time 1.1.2(29), 1.1.5(4)
run-time 1.1.2(30), 1.1.5(6), 11.5(2/3), 11.6(1/3)
See also bounded error, erroneous execution
ESA
in Ada.Characters.Latin_1 A.3.3(17)
ESC
in Ada.Characters.Latin_1 A.3.3(6)
Establish_RPC_Receiver
in System.RPC E.5(12)
ETB
in Ada.Characters.Latin_1 A.3.3(6)
ETX
in Ada.Characters.Latin_1 A.3.3(5)
evaluated operative constituent 4.4(9.2/5)
evaluation 1.3.4(5/5), 3.1(11)
aggregate 4.3(5)
allocator 4.8(7/2)
array_aggregate 4.3.3(21/5)
attribute_reference 4.1.4(11/5)
case_expression 4.5.7(21/3)
concatenation 4.5.3(5)
dereference 4.1(13)
discrete_range 3.6.1(8)
extension_aggregate 4.3.2(7)
generalized_reference 4.1.5(8/3)
generic_association 12.3(21)
generic_association for a formal object of mode in 12.4(11)
if_expression 4.5.7(20/3)
indexed_component 4.1.1(7)
initialized allocator 4.8(7/2)
membership test 4.5.2(27/4)
name 4.1(11/2)
name that has a prefix 4.1(12)
null literal 4.2(9/5)
numeric literal 4.2(9/5)
parameter_association 6.4.1(7)
prefix 4.1(12)
primary that is a name 4.4(10)
qualified_expression 4.7(4/5)
quantified_expression 4.5.8(6/5)
range 3.5(9)
range_attribute_reference 4.1.4(11/5)
record_aggregate 4.3.1(18)
record_component_association_list 4.3.1(19/5)
reduction_attribute_reference 4.5.10(24/6)
selected_component 4.1.3(14)
short-circuit control form 4.5.1(7)
slice 4.1.2(7)
string_literal 4.2(10/6)
uninitialized allocator 4.8(8)
Val 3.5.5(7), K.2(261)
Value 3.5(55/5)
value conversion 4.6(28)
value_sequence 4.5.10(20/6)
view conversion 4.6(52)
Wide_Value 3.5(43/5)
Wide_Wide_Value 3.5(39.4/3)
exception 1.3.5(2/5), 11(1/3), 11.1(1)
exception occurrence 1.3.5(3/5), 11(1/3)
exception_choice 11.2(5)
used 11.2(3/5), P.1
exception_declaration 11.1(2/3)
used 3.1(3/3), P.1
exception_handler 11.2(3/5)
used 11.2(2), P.1
Exception_Id
in Ada.Exceptions 11.4.1(2/5)
Exception_Identity
in Ada.Exceptions 11.4.1(5/2)
Exception_Information
in Ada.Exceptions 11.4.1(5/2)
Exception_Message
in Ada.Exceptions 11.4.1(4/3)
Exception_Name
in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Exception_Occurrence
in Ada.Exceptions 11.4.1(3/5)
Exception_Occurrence_Access
in Ada.Exceptions 11.4.1(3/5)
exception_renaming_declaration 8.5.2(2/3)
used 8.5(2), P.1
Exceptions
child of Ada 11.4.1(2/5)
Exchange
child of System.Atomic_Operations C.6.2(5/5)
Exchange_Handler
in Ada.Interrupts C.3.2(8)

Exclamation
  in Ada.Characters.Latin_1 A.3.3(8)
exclamation point 2.1(15/6)
Exclude
  in Ada.Containers.Hashed_Maps A.18.5(24/5)
  in Ada.Containers.Hashed_Sets A.18.8(23/5), A.18.8(54/5)
  in Ada.Containers.Ordered_Maps A.18.6(23/5)
  in Ada.Containers.Ordered_Sets A.18.9(22/5), A.18.9(67/5)
excluded from stable property checks 7.3.4(24/5)
excludes null
  subtype 3.10(13.1/2)
exclusive
  protected operation 9.5.1(2.4/4)
Exclusive_Functions aspect 9.5.1(2.2/4)
executable pragma
  [*partial*] 2.8(7.2/6)
  Assert 11.4.2(4/6)
  Inspection_Point H.3.2(4/6)
execution 1.3.4(7/5), 3.1(11)
  abort_statement 9.8(4)
  aborting the execution of a construct 9.8(5)
  accept_statement 9.5.2(24/5)
  Ada program 9(1/5)
  assignment_statement 5.2(7), 7.6(17), 7.6.1(12/2)
  asynchronous_select with a delay_statement trigger 9.7.4(7)
  asynchronous_select with a procedure call trigger 9.7.4(6/2)
  asynchronous_select with an entry call trigger 9.7.4(6/2)
  block_statement 5.6(5)
  call on a dispatching operation 3.9.2(14)
  call on an inherited subprogram 3.4(27/6)
  case_statement 5.4(11/3)
  conditional_entry_call 9.7.3(3)
  delay_statement 9.6(20)
  dynamically enclosing 11.4(2)
  entry_body 9.5.2(26)
  entry_call_statement 9.5.3(8)
  exit_statement 5.7(5)
  extended_return_statement 6.5(5.12/5)
  goto_statement 5.8(5)
  handled_sequence_of_statements 11.2(10)
  handler 11.4(7)
  if_statement 5.3(5/3)
  instance of Unchecked_Deallocation 7.6.1(10)
  loop_statement 5.5(7/5)
  loop_statement with a for loop_parameter_specification 5.5(9/5)
  loop_statement with a while iteration_scheme 5.5(8)
  null_statement 5.1(13)
  partition 10.2(25)
  pragma 2.8(12/6)
  program 10.2(25)
  protected subprogram call 9.5.1(3)
  raise_statement with an exception_name 11.3(4/4)
  re-raise statement 11.3(4/4)
  remote subprogram call E.4(9)
  requeue protected entry 9.5.4(9)
  requeue task entry 9.5.4(8)
  requeue_statement 9.5.4(7/5)
  selective_accept 9.7.1(15)
  sequence_of_statements 5.1(15)
  simple_return_statement 6.5(6/2)
  subprogram call 6.4(10/2)
  subprogram_body 6.3(7)
  task 9.2(1)
  task_body 9.2(1)
  timed_entry_call 9.7.2(4/2)
execution resource
  associated with a protected object 9.4(18)
  required for a task to run 9(10/5)
execution time
  of a task D.14(11/3)
Execution_Time
  child of Ada D.14(3/5)
exhaust
  a budget D.14.2(14/2)
exist
  cease to 7.6.1(11/6), 13.11.2(10/4), 13.11.5(7.1/4)
Exists
  in Ada.Directories A.16(24/2)
  in Ada.Environment_Variables A.17(5/2)
exit_statement 5.7(2)
  used 5.1(4/2), P.1
Exit_Status
  in Ada.Command_Line A.15(7)
Exp
  in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
  in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
expanded name 4.1.3(4)
Expanded_Name
  in Ada.Tags 3.9(7/2)
expected profile 8.6(26)
  accept_statement entry_direct_name 9.5.2(11)
  Access attribute_reference prefix 3.10.2(2.3/2)
  attribute_definition_clause name 13.3(4)
  character_literal 4.2(3)
  formal subprogram actual 12.6(6)
  formal subprogram default_name 12.6(5)
  name in an aspect_specification 13.1.1(8/6)
  subprogram_renaming_declaration 8.5.4(3)
expected type 8.6(20/2)
  abort_statement task_name 9.8(3/5)
  access attribute_reference 3.10.2(2/2)
  Access attribute_reference prefix 3.10.2(2.3/2)
  actual parameter 6.4.1(3)
  aggregate 4.3(3/6)
  allocator 4.8(3/3)
  array delta aggregate 4.3.4(6/5)
  array_aggregate 4.3.3(7/2)
  array_aggregate component expression 4.3.3(7/2)
  array_aggregate discrete_choice 4.3.3(8)
  assignment_statement expression 5.2(4/6)
  assignment_statement variable_name 5.2(4/6)
  Attach_Handler pragma second argument J.15.7(6/3)
  attribute_definition_clause expression or name 13.3(4)
  attribute_designator expression 4.1.4(7)
  base expression of a delta aggregate 4.3.4(7/5)
  case_expression selecting_expression 4.5.7(15/3)
  case_expression_alternative discrete_choice 4.5.7(15/3)
  case_statement selecting_expression 5.4(4/3)
  case_statement_alternative discrete_choice 5.4(4/3)
  character_literal 4.2(3)
  code_statement 13.8(4)
  component_clause expressions 13.5.1(7)
  component_declaration default_expression 3.8(7)
  condition 4.5.7(14/3)
  CPU pragma argument J.15.9(3/3)
  decimal fixed point type digits 3.5.9(6)
  delay_relative_statement expression 9.6(5)
  delay_until_statement expression 9.6(5)
  delta aggregate base expression 4.3.4(7/5)
  delta_constraint expression J.3(3/4)

dependent_expression 4.5.7(8/3)
dereference name 4.1(8)
discrete_subtype_definition range 3.6(8)
discriminant default_expression 3.7(7)
discriminant_association expression 3.7.1(6)
Dispatching_Domains pragma argument J.15.10(3/3)
entry_index 9.5.2(11)
enumeration_representation_clause expressions 13.4(4)
expression in an aspect_specification 13.1.1(7/3)
expression of a Default_Component_Value aspect 3.6(22.4/3)
expression of a Default_Value aspect 3.5(56.5/3)
expression of a predicate aspect 3.2.4(2/3)
expression of expression function 6.8(3/4)
expression of extended_return_object_declaration 6.5(3/5)
expression of simple_return_statement 6.5(3/5)
extension_aggregate 4.3.2(4/2)
extension_aggregate ancestor expression 4.3.2(4/2)
external name J.15.5(6/3)
first_bit 13.5.1(7)
fixed point type delta 3.5.9(6)
generic formal in object actual 12.4(4)
generic formal object default_expression 12.4(3)
has an Aggregate aspect 4.3.5(22/6)
index_constraint discrete_range 3.6.1(4)
indexable_container_object_prefix 4.1.6(11/3)
indexed_component expression 4.1.1(4)
integer_literal 4.2(4/6)
Interrupt_Priority pragma argument J.15.11(5/3)
invariant expression 7.3.2(4/3)
iterable_name 5.5.2(3/3)
iterator_name 5.5.2(3/3)
last_bit 13.5.1(7)
link name J.15.5(6/3)
linker options B.1(10.1/3)
membership test simple_expression 4.5.2(3/3)
modular_type_definition expression 3.5.4(5/5)
name in an aspect_specification 13.1.1(7/3)
number_declaration expression 3.3.2(3)
object in an aspect_specification 13.1.1(6/3)
object_declaration initialization expression 3.3.1(4)
parameter default_expression 6.1(17)
position 13.5.1(7)
postcondition expression 6.1.1(6/3)
precondition expression 6.1.1(6/3)
Priority pragma argument J.15.11(5/3)
quantified_expression 4.5.8(5/3)
range simple_expressions 3.5(5)
range_attribute_designator expression 4.1.4(7)
range_constraint range 3.5(5)
real_literal 4.2(4/6)
real_range_specification bounds 3.5.7(5/5)
record delta aggregate 4.3.4(5/5)
record_aggregate 4.3.1(8/2)
record_component_association expression 4.3.1(10)
reference_object_name 4.1.5(5/3)
Relative_Deadline pragma argument J.15.12(3/3)
requested decimal precision 3.5.7(4)
restriction parameter expression 13.12(5)
selecting_expression case_expression 4.5.7(15/3)
selecting_expression case_statement 5.4(4/3)
short-circuit control form relation 4.5.1(1)
signed_integer_type_definition simple_expression 3.5.4(5/5)
slice discrete_range 4.1.2(4)
Storage_Size pragma argument J.15.4(4/3)
string_literal 4.2(4/6)
subpool_handle_name 4.8(3/3)
type_conversion operand 4.6(6)
variant_part discrete_choice 3.8.1(6)
expiration time
[*partial*] 9.6(1)
for a delay_relative_statement 9.6(20)
for a delay_until_statement 9.6(20)
expires
execution timer D.14.1(15/3)
explicit class-wide postcondition expression 7.3.4(23/5)
explicit declaration 3.1(5)
explicit initial value 3.3.1(1/3)
explicit specific postcondition expression 7.3.4(23/5)
explicit_actual_parameter 6.4(6)
used 6.4(5), P.1
explicit_dereference 4.1(5)
used 4.1(2/5), P.1
explicit_generic_actual_parameter 12.3(5)
used 12.3(4), P.1
explicitly aliased parameter 6.1(23.1/3)
explicitly assign 10.2(2/5)
explicitly limited record 3.8(13.1/3)
exponent 2.4.1(4), 4.5.6(11/3)
used 2.4.1(2), 2.4.2(2), P.1
Exponent attribute A.5.3(18)
exponentiation operator 4.5(1), 4.5.6(7)
Export aspect B.1(1/3)
Export pragma J.15.5(3/3), L(13.1/3)
exported entity B.1(23/3)
expression 4.4(1/3), 4.4(2)
case 4.5.7(1/3)
conditional 4.5.7(1/3)
declare 4.5.9(1/5)
if 4.5.7(1/3)
of an extended_return_statement 6.5(5.1/5)
parenthesized 4.4(8.1/6), 4.4(9.1/6)
predicate-static 3.2.4(15/3)
quantified 4.5.8(0.1/4)
reduction 4.5.10(1/5)
used 2.8(3/3), 3.3.1(2/3), 3.3.2(2), 3.5.4(4), 3.5.7(2), 3.5.9(3), 3.5.9(4), 3.7(6), 3.7.1(3/5), 4.1.1(2), 4.1.4(3/2), 4.1.4(5), 4.3.1(4/2), 4.3.2(3), 4.3.3(3/5), 4.3.3(5.1/5), 4.3.3(5/5), 4.3.4(3/5), 4.3.4(4/5), 4.3.5(15/5), 4.3.5(18/5), 4.3.5(20/5), 4.3.5(21/5), 4.4(7/5), 4.5.7(3/3), 4.5.7(4/3), 4.5.7(5/3), 4.5.7(6/3), 4.5.8(3/3), 4.5.9(2/5), 4.5.10(5/6), 4.6(2), 4.7(2), 5.2(2), 5.4(2/3), 6.4(6), 6.5(2.1/5), 6.5(2/2), 6.8(2/4), 9.5.2(4), 9.6(3), 9.6(4), 11.3(2/2), 11.4.2(3/2), 12.3(5), 13.1.1(4/5), 13.3(2), 13.5.1(4), 13.12(4.1/2), B.1(8), B.1(10.1/3), D.2.2(2.2/2), J.7(1), J.8(1), J.15.4(2/3), J.15.5(2/3), J.15.5(3/3), J.15.7(4/3), J.15.9(2/3), L(2.2/2), L(6.1/3), L(8.2/3), L(13.1/3), L(14.1/3), L(19), L(27.2/2), L(35.1/3), P.1
expression function 6.8(3.1/5)
static 6.8(3.6/5)
expression_function_declaration 6.8(2/4)
used 3.1(3/3), 9.4(8/4), P.1
extended_digit 2.4.2(5)
used 2.4.2(4), P.1
extended_global_mode H.7(3/5)
used 6.1.2(4/5), P.1
Extended_Index *subtype of* Index_Type'Base
in Ada.Containers.Vectors A.18.2(7/2)
extended_return_object_declaration 6.5(2.1/5)
used 6.5(2.2/3), P.1
extended_return_statement 6.5(2.2/3)
used 5.1(5/5), P.1

extension
of a private type 3.9(2.1/2), 3.9.1(1/2)
of a record type 3.9(2.1/2), 3.9.1(1/2)
of a type 3.9(2/2), 3.9.1(1/2)
in Ada.Directories A.16(18/5)
extension_aggregate 4.3.2(2)
used 4.3(2/5), P.1
external call 9.5(4/3)
external effect
of the execution of an Ada program 1.1.3(8)
volatile/atomic objects C.6(20/5)
external file A.7(1)
external interaction 1.1.3(8)
external name B.1(34)
external requeue 9.5(7)
external streaming
type supports 13.13.2(52/3)
External_Name aspect B.1(1/3)
External_Tag
in Ada.Tags 3.9(7/2)
External_Tag aspect 13.3(75/3), K.2(65)
External_Tag attribute 13.3(75/3)
External_Tag clause 13.3(7/2), 13.3(75/3), K.2(65)
extra permission to avoid raising exceptions 11.6(5/5)
extra permission to reorder actions 11.6(6/5)

F

factor 4.4(6)
used 4.4(5), P.1
factory 3.9(30/2)
failure
of a language-defined check 11.5(2/3)
in Ada.Command_Line A.15(8)
fall-back handler C.7.3(9/2)
False 3.5.3(1)
family
entry 9.5.2(20)
Feminine_Ordinal_Indicator
in Ada.Characters.Latin_1 A.3.3(21/3)
FF
in Ada.Characters.Latin_1 A.3.3(5)
Field *subtype of* Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
in Ada.Text_IO A.10.1(6)
FIFO_Queuing queuing policy D.4(7/5)
FIFO_Within_Priorities task dispatching policy D.2.3(2/2)
file
as file object A.7(2/3)
file name A.16(46/2)
file terminator A.10(7)
File_Access
in Ada.Text_IO A.10.1(18)
File_Kind
in Ada.Directories A.16(22/2)
File_Mode
in Ada.Direct_IO A.8.4(4)
in Ada.Sequential_IO A.8.1(4)
in Ada.Streams.Stream_IO A.12.1(6)
in Ada.Text_IO A.10.1(4)
File_Size
in Ada.Directories A.16(23/2)
File_Type
in Ada.Direct_IO A.8.4(3)
in Ada.Sequential_IO A.8.1(3)
in Ada.Streams.Stream_IO A.12.1(5/5)
in Ada.Text_IO A.10.1(3)
filter
iterator construct 5.5(6.2/5)
Filter_Type
in Ada.Directories A.16(30/2)
finalization
of a master 7.6.1(4)
of a protected object 9.4(20)
of a protected object C.3.1(12/3)
of a task object J.7.1(8)
of an object 7.6.1(5)
of environment task for a foreign language main subprogram B.1(39/3)
child of Ada 7.6(4/5)
Finalize 7.6(2)
in Ada.Finalization 7.6(6/2), 7.6(8/2)
Find
in Ada.Containers.Doubly_Linked_Lists A.18.3(41/5)
in Ada.Containers.Hashed_Maps A.18.5(30/5)
in Ada.Containers.Hashed_Sets A.18.8(43/5), A.18.8(56/6)
in Ada.Containers.Multiway_Trees A.18.10(38/5)
in Ada.Containers.Ordered_Maps A.18.6(38/5)
in Ada.Containers.Ordered_Sets A.18.9(49/5), A.18.9(69/5)
in Ada.Containers.Vectors A.18.2(68/5)
Find_In_Subtree
in Ada.Containers.Multiway_Trees A.18.10(39.1/5), A.18.10(39/5)
Find_Index
in Ada.Containers.Vectors A.18.2(67/2)
Find_Token
in Ada.Strings.Bounded A.4.4(50.1/3), A.4.4(51)
in Ada.Strings.Fixed A.4.3(15.1/3), A.4.3(16)
in Ada.Strings.Unbounded A.4.5(45.1/3), A.4.5(46)
Fine_Delta
in System 13.7(9)
First
in Ada.Containers.Doubly_Linked_Lists A.18.3(33/5)
in Ada.Containers.Hashed_Maps A.18.5(27/5)
in Ada.Containers.Hashed_Sets A.18.8(40/5)
in Ada.Containers.Ordered_Maps A.18.6(28/5)
in Ada.Containers.Ordered_Sets A.18.9(41/5)
in Ada.Containers.Vectors A.18.2(58/5)
in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.6/5)
First attribute 3.5(12), 3.6.2(3)
first element
of a hashed set A.18.8(68/2)
of a set A.18.7(6/2)
of an ordered set A.18.9(81/3)
first node
of a hashed map A.18.5(46/2)
of a map A.18.4(6/2)
of an ordered map A.18.6(58/3)
first subtype 3.2.1(6/5), 3.4.1(5)
First(N) attribute 3.6.2(4)
first_bit 13.5.1(5)
used 13.5.1(3), P.1
First_Bit attribute 13.5.2(3/2)
First_Child
in Ada.Containers.Multiway_Trees A.18.10(60.1/5), A.18.10(60/5)
First_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(61.1/5), A.18.10(61/5)
First_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(34/5)
in Ada.Containers.Ordered_Maps A.18.6(29/5)
in Ada.Containers.Ordered_Sets A.18.9(42/5)
in Ada.Containers.Vectors A.18.2(59/5)
First_Index
in Ada.Containers.Vectors A.18.2(57/5)
First_Key
in Ada.Containers.Ordered_Maps A.18.6(30/5)
First_Valid attribute 3.5.5(7.2/4)
Fixed
child of Ada.Strings A.4.3(5/5)
fixed point
has intergral values A.5.4(5/6)
fixed point type 3.5.9(1)
Fixed_Conversions
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(14/5)

Fixed_IO
  in Ada.Text_IO A.10.1(68)
fixed_point_definition 3.5.9(2)
  used 3.5.6(2), P.1
Float 3.5.7(12), 3.5.7(14)
  in Standard A.1(21)
Float_Conversions
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5)
Float_IO
  in Ada.Text_IO A.10.1(63)
Float_Random
  child of Ada.Numerics A.5.2(5/5)
Float_Text_IO
  child of Ada A.10.9(33)
Float_Wide_Text_IO
  child of Ada A.11(2/2)
Float_Wide_Wide_Text_IO
  child of Ada A.11(3/2)
Floating
  in Interfaces.COBOL B.4(9)
floating point type 3.5.7(1)
floating_point_definition 3.5.7(2)
  used 3.5.6(2), P.1
Floor
  in Ada.Containers.Ordered_Maps A.18.6(40/5)
  in Ada.Containers.Ordered_Sets A.18.9(50/2), A.18.9(70/5)
Floor attribute A.5.3(30), A.5.4(8/6)
Flush
  in Ada.Direct_IO A.8.4(10.1/5)
  in Ada.Sequential_IO A.8.1(10.1/5)
  in Ada.Streams.Stream_IO A.12.1(25/1)
  in Ada.Text_IO A.10.1(21/5)
Fore attribute 3.5.10(4)
form
  of an external file A.7(1)
  in Ada.Direct_IO A.8.4(9), A.8.4(18.13/5), A.8.4(18.6/5)
  in Ada.Sequential_IO A.8.1(9), A.8.1(15.13/5), A.8.1(15.6/5)
  in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.13/5), A.12.1(26.6/5)
  in Ada.Text_IO A.10.1(12), A.10.1(85.13/5), A.10.1(85.6/5)
formal object, generic 12.4(1)
formal package, generic 12.7(1)
formal parameter
  of a subprogram 6.1(17)
formal parameter set H.7.1(13/5)
formal subprogram, generic 12.6(1)
formal subtype 12.5(5)
formal type 12.5(5)
formal_abstract_subprogram_declaration 12.6(2.2/3)
  used 12.6(2/2), P.1
formal_access_type_definition 12.5.4(2)
  used 12.5(3/2), P.1
formal_array_type_definition 12.5.3(2)
  used 12.5(3/2), P.1
formal_complete_type_declaration 12.5(2.1/5)
  used 12.5(2/3), P.1
formal_concrete_subprogram_declaration 12.6(2.1/3)
  used 12.6(2/2), P.1
formal_decimal_fixed_point_definition 12.5.2(7)
  used 12.5(3/2), P.1
formal_derived_type_definition 12.5.1(3/5)
  used 12.5(3/2), P.1
formal_discrete_type_definition 12.5.2(2)
  used 12.5(3/2), P.1
formal_floating_point_definition 12.5.2(5)
  used 12.5(3/2), P.1
formal_group_designator H.7.1(5/5)
  used H.7.1(4/5), P.1
formal_incomplete_type_declaration 12.5(2.2/5)
  used 12.5(2/3), P.1
formal_interface_type_definition 12.5.5(2/2)
  used 12.5(3/2), P.1
formal_modular_type_definition 12.5.2(4)
  used 12.5(3/2), P.1
formal_object_declaration 12.4(2/3)
  used 12.1(6), P.1
formal_ordinary_fixed_point_definition 12.5.2(6)
  used 12.5(3/2), P.1
formal_package_actual_part 12.7(3/2)
  used 12.7(2/3), P.1
formal_package_association 12.7(3.1/2)
  used 12.7(3/2), P.1
formal_package_declaration 12.7(2/3)
  used 12.1(6), P.1
formal_parameter_name H.7.1(6/5)
  used H.7.1(4/5), P.1
formal_parameter_set H.7.1(4/5)
formal_part 6.1(14)
  used 5.5.3(3/5), 6.1(12), 6.1(13/2), P.1
formal_private_type_definition 12.5.1(2)
  used 12.5(3/2), P.1
formal_signed_integer_type_definition 12.5.2(3)
  used 12.5(3/2), P.1
formal_subprogram_declaration 12.6(2/2)
  used 12.1(6), P.1
formal_type_declaration 12.5(2/3)
  used 12.1(6), P.1
formal_type_definition 12.5(3/2)
  used 12.5(2.1/5), P.1
format_effector 2.1(13/3)
Formatting
  child of Ada.Calendar 9.6.1(15/5)
Fortran
  calling convention B.5(1/3)
  child of Interfaces B.5(4/5)
Fortran interface B.5(1/3)
Fortran standard 1.2.1(3/5)
Fortran_Character
  in Interfaces.Fortran B.5(12/3)
Fortran_Integer
  in Interfaces.Fortran B.5(5)
forward iterator 5.5.2(4/5)
Forward_Iterator
  in Ada.Iterator_Interfaces 5.5.1(3/3)
Fraction attribute A.5.3(21)
Fraction_One_Half
  in Ada.Characters.Latin_1 A.3.3(22)
Fraction_One_Quarter
  in Ada.Characters.Latin_1 A.3.3(22)
Fraction_Three_Quarters
  in Ada.Characters.Latin_1 A.3.3(22)
Free
  in Ada.Strings.Unbounded A.4.5(7)
  in Interfaces.C.Strings B.3.1(11)
freed
  See nonexistent 13.11.2(10/4)
freeing storage 13.11.2(1)
freezing
  by a constituent of a construct 13.14(4/1)
  by an expression 13.14(8/4)
  by an implicit call 13.14(8.1/3)
  by an object name 13.14(8/4)
  class-wide type caused by the freezing of the specific type 13.14(15/5)
  constituents of a full type definition 13.14(15/5)
  designated subtype caused by an allocator 13.14(13)
  entity 13.14(2)
  entity caused by a body 13.14(3/5)
  entity caused by a construct 13.14(4/1)
  entity caused by a name 13.14(11)
  entity caused by the end of an enclosing construct 13.14(3/5)
  expression of an expression function by a call 13.14(10.1/4)
  expression of an expression function by Access attribute 13.14(10.3/4)
  expression of an expression function by an instantiation 13.14(10.2/4)
  expression_function_declaration 13.14(5.1/4)
  first subtype caused by the freezing of the type 13.14(15/5)
  generic_instantiation 13.14(5/3)
  nominal subtype caused by a name 13.14(11)
  object_declaration 13.14(6)

profile 13.14(2.1/3)
profile of a callable entity by an instantiation 13.14(10.2/4)
profile of a function call 13.14(10.1/4)
progenitor types 13.14(7.1/2)
specific type caused by the freezing of the class-wide type 13.14(15/5)
subtype caused by a record extension 13.14(7)
subtype caused by an implicit conversion 13.14(8.2/1)
subtype caused by an implicit dereference 13.14(11.1/1)
subtypes of the profile of a callable entity 13.14(14/3)
type caused by a range 13.14(12)
type caused by an expression 13.14(10/5)
type caused by the freezing of a subtype 13.14(15/5)
freezing points
entity 13.14(2)
Friday
in Ada.Calendar.Formatting 9.6.1(17/2)
From_Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5), A.5.6(13/5)
From_Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(13/5), A.5.7(14.1/6), A.5.7(14/5)
From_Quotient_String
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
From_String
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(15/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(16/5)
From_Universal_Image
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(16/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(17/5), A.5.7(18/5)
FS
in Ada.Characters.Latin_1 A.3.3(6)
full access
object C.6(8.2/5)
type C.6(8.2/5)
full conformance
for discrete_subtype_definitions 6.3.1(24)
for expressions 6.3.1(19)
for known_discriminant_parts 6.3.1(23)
for profiles 6.3.1(18/3)
required 3.10.1(4/3), 6.3(4/5), 6.7(2.1/3), 6.8(4/3), 7.3(9/5), 8.3(12.3/5), 8.5.4(5/3), 9.5.2(14), 9.5.2(16), 9.5.2(17), 10.1.3(11/5), 10.1.3(12)
full constant declaration 3.3.1(6/5)
corresponding to a formal object of mode in 12.4(10/2)
full declaration 7.4(2/3)
full name
of a file A.16(47/2)
full stop 2.1(15/6)
full type 1.3.1(32/5), 3.2.1(8/2)
full type definition 3.2.1(8/2)
full view 1.3.1(33/5)
of a type 3.2.1(8/2)
Full_Access_Only aspect C.6(6.5/5)
Full_Name
in Ada.Directories A.16(15/5), A.16(39/2)
Full_Stop
in Ada.Characters.Latin_1 A.3.3(8)
full_type_declaration 3.2.1(3/3)
used 3.2.1(2), P.1
function 1.3.2(1/5), 6(1)
constructor 3.2.3(8/6)
expression 6.8(3.1/5)
property 7.3.4(2/5)
stable property 7.3.4(7/5)
with a controlling access result 3.9.2(2/3)
with a controlling result 3.9.2(2/3)
function call
master of 3.10.2(10.1/3)
function instance 12.3(13)
function_call 6.4(3)
used 4.1(2/5), P.1
function_specification 6.1(4.2/2)
used 6.1(4/2), 6.8(2/4), P.1

G

general access type 3.10(7/1), 3.10(8)
general_access_modifier 3.10(4)
used 3.10(3), P.1
generalized iterator 5.5.2(3/3)
generalized_indexing 4.1.6(10/3)
used 4.1(2/5), P.1
generalized_reference 4.1.5(4/3)
used 4.1(2/5), P.1
Generate_Deadlines pragma D.2.6(9.4/5), L(13.2/5)
generation
of an interrupt C.3(2)
Generator
in Ada.Numerics.Discrete_Random A.5.2(19)
in Ada.Numerics.Float_Random A.5.2(7)
generic actual 12.3(7/5)
generic actual parameter 12.3(7/5)
generic actual subtype 12.5(4)
generic actual type 12.5(4)
generic body 12.2(1)
generic contract issue 10.2.1(10/2)
[*partial*] 3.2.4(29/3), 3.4(5.1/3), 3.7(10/3), 3.7.1(7/3), 3.9.1(3/2), 3.9.3(7/5), 3.9.4(17/2), 3.10.2(28.1/3), 3.10.2(32/5), 4.1.6(9/5), 4.2.1(16/5), 4.3.5(11.3/6), 4.5.2(9.8/6), 4.6(24.28/6), 4.8(5.6/3), 4.9(37/2), 6.1.1(17.2/4), 6.4.1(5.4/4), 6.4.1(6.3/4), 6.5.1(6/5), 7.3(8), 8.3(26/6), 8.3.1(7/2), 8.5.1(5.1/5), 8.5.4(4.3/2), 9.1(9.9/2), 9.4(11.13/2), 9.4(11.8/2), 9.5(17/3), 9.5(71/5), 9.5.2(13.4/2), 10.2.1(11.20/6), 10.2.1(11/5), 10.2.1(17/5), 12.4(8.3/5), 12.6(8.3/2), 13.1.1(18.8/6), 13.11.2(3.1/3), 13.11.4(23/3), 13.13.2(49/4), B.3.3(10/3), C.3.1(7/3), D.2.1(1.12/5), E.2.3(15.1/5), H.4.1(4/5), J.15.7(7/3), J.15.14(12/5)
generic formal 12.1(9)
generic formal object 12.4(1)
generic formal package 12.7(1)
generic formal subprogram 12.6(1)
generic formal subtype 12.5(5)
generic formal type 12.5(5)
generic function 12.1(8/2)
generic instance 1.3.3(12/5)
generic package 12.1(8/2)
generic procedure 12.1(8/2)
generic subprogram 12.1(8/2)
generic unit 1.3.3(13/5), 12(1)
See also dispatching operation 3.9(1)
generic_actual_part 12.3(3)
used 12.3(2/3), 12.7(3/2), P.1
Generic_Array_Sort
child of Ada.Containers A.18.26(3/5)
generic_association 12.3(4)
used 12.3(3), 12.7(3.1/2), P.1
Generic_Bounded_Length
in Ada.Strings.Bounded A.4.4(4)
Generic_Complex_Arrays
child of Ada.Numerics G.3.2(2/5)
Generic_Complex_Elementary_Functions
child of Ada.Numerics G.1.2(2/5)
Generic_Complex_Types
child of Ada.Numerics G.1.1(2/5)
Generic_Constrained_Array_Sort
child of Ada.Containers A.18.26(7/5)

generic_declaration 12.1(2)
used 3.1(3/3), 10.1.1(5), P.1
Generic_Dispatching_Constructor
child of Ada.Tags 3.9(18.2/5)
Generic_Elementary_Functions
child of Ada.Numerics A.5.1(3/5)
generic_formal_parameter_declaration 12.1(6)
used 12.1(5), P.1
generic_formal_part 12.1(5)
used 12.1(3/3), 12.1(4), P.1
generic_instantiation 12.3(2/3)
used 3.1(3/3), 10.1.1(5), P.1
Generic_Keys
in Ada.Containers.Hashed_Sets A.18.8(50/5)
in Ada.Containers.Ordered_Sets A.18.9(62/5)
generic_package_declaration 12.1(4)
used 12.1(2), P.1
Generic_Real_Arrays
child of Ada.Numerics G.3.1(2/5)
generic_renaming_declaration 8.5.5(2/3)
used 8.5(2), 10.1.1(6), P.1
Generic_Sort
child of Ada.Containers A.18.26(9.2/5)
Generic_Sorting
in Ada.Containers.Doubly_Linked_-Lists A.18.3(47/5)
in Ada.Containers.Vectors A.18.2(75/5)
generic_subprogram_declaration 12.1(3/3)
used 12.1(2), P.1
Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(20/5)
in Ada.Text_IO A.10.1(41/5), A.10.1(47/5), A.10.1(54/5), A.10.1(55/5), A.10.1(59/5), A.10.1(60/5), A.10.1(65/5), A.10.1(67/5), A.10.1(70/5), A.10.1(72/5), A.10.1(75/5), A.10.1(77/5), A.10.1(81/5), A.10.1(83/5)
in Ada.Text_IO.Complex_IO G.1.3(6), G.1.3(8/5)
Get_CPU
in Ada.Interrupts C.3.2(10.1/3)
in System.Multiprocessors.Dispatching_Domains D.16.1(13/3)
Get_CPU_Set
in System.Multiprocessors.Dispatching_Domains D.16.1(9.3/4)
Get_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Get_Dispatching_Domain
in System.Multiprocessors.Dispatching_Domains D.16.1(10/3)
Get_First_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(8/3)
Get_Immediate
in Ada.Text_IO A.10.1(44/5), A.10.1(45/5)
Get_Last_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(9/4)
Get_Last_Release_Time
in Ada.Dispatching.EDF D.2.6(9/5)
Get_Line
in Ada.Text_IO A.10.1(49.1/5), A.10.1(49/5)
in Ada.Text_IO.Bounded_IO A.10.11(8/2), A.10.11(9/2), A.10.11(10/2), A.10.11(11/2)
in Ada.Text_IO.Unbounded_IO A.10.12(8/2), A.10.12(9/2), A.10.12(10/2), A.10.12(11/2)
Get_Next_Entry
in Ada.Directories A.16(35/2)
Get_Priority
in Ada.Dynamic_Priorities D.5.1(5)
Get_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Get_UTF_8
in Ada.Strings.Text_Buffers.Unbounded A.4.12(23/5)
Global aspect 6.1.2(11/5), 6.1.2(16/5)
global to 8.1(15)
global variable set 6.1.2(24/5)
Global'Class aspect 6.1.2(15/5)
global_aspect_definition 6.1.2(2/5)
used 13.1.1(4/5), P.1
global_aspect_element 6.1.2(3/5)
used 6.1.2(2/5), P.1
global_designator 6.1.2(7/5)
used 6.1.2(2/5), P.1
global_mode 6.1.2(4/5)
used 6.1.2(2/5), 6.1.2(3/5), P.1
global_name 6.1.2(8/5)
used 6.1.2(6/5), 6.1.2(7/5), P.1
global_set 6.1.2(6/5)
used 6.1.2(3/5), P.1
goto_statement 5.8(2)
used 5.1(4/2), P.1
govern a variant 3.8.1(20)
govern a variant_part 3.8.1(20)
grammar
complete listing P.1(1/5)
cross reference P.2(1/3)
notation 1.1.4(3)
resolution of ambiguity 8.6(3)
under Syntax heading 1.1.2(25)
graphic character
a category of Character A.3.2(23)
graphic_character 2.1(14/3)
used 2.5(2), 2.6(3), P.1
Graphic_Set
in Ada.Strings.Maps.Constants A.4.6(4)
Grave
in Ada.Characters.Latin_1 A.3.3(13)
greater than operator 4.5(1), 4.5.2(1)
greater than or equal operator 4.5(1), 4.5.2(1)
greater-than sign 2.1(15/6)
Greater_Than_Sign
in Ada.Characters.Latin_1 A.3.3(10)
Greatest_Common_Divisor
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(19/5)
Group_Budget
in Ada.Execution_Time.Group_Budgets D.14.2(4/3)
Group_Budget_Error
in Ada.Execution_Time.Group_Budgets D.14.2(11/2)
Group_Budget_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(5/5)
Group_Budgets
child of Ada.Execution_Time D.14.2(3/5)
GS
in Ada.Characters.Latin_1 A.3.3(6)
guard 9.7.1(3)
used 9.7.1(2), P.1

H

handle
an exception 11(1/3)
an exception occurrence 11.4(1), 11.4(7)
subpool 13.11.4(18/3)
handle an exception 1.3.5(4/5)
handled_sequence_of_statements 11.2(2)
used 5.6(2), 6.3(2/3), 6.5(2.2/3), 7.2(2/3), 9.1(6/3), 9.5.2(3), 9.5.2(5/5), P.1
handler
execution timer D.14.1(13/2)
group budget D.14.2(14/2)
interrupt C.3(2)
termination C.7.3(8/3)
timing event D.15(10/2)

Handling
child of Ada.Characters A.3.2(2/5)
child of Ada.Wide_Characters A.3.5(3/5)
child of Ada.Wide_Wide_Characters A.3.6(1/3)
has intergral values A.5.4(5/6)
Has_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.1/5), A.18.3(9.2/5), A.18.3(51.6/5), A.18.3(69.4/5)
in Ada.Containers.Hashed_Maps A.18.5(6.1/5), A.18.5(6.2/5), A.18.5(37.7/5)
in Ada.Containers.Hashed_Sets A.18.8(6.1/5), A.18.8(6.2/5), A.18.8(59.6/5)
in Ada.Containers.Multiway_Trees A.18.10(12.1/5), A.18.10(12/5), A.18.10(70.6/5), A.18.10(92.1/5)
in Ada.Containers.Ordered_Maps A.18.6(7.1/5), A.18.6(7.2/5), A.18.6(51.8/5)
in Ada.Containers.Ordered_Sets A.18.9(7.1/5), A.18.9(7.2/5), A.18.9(74.6/5)
in Ada.Containers.Vectors A.18.2(11.1/5), A.18.2(11.2/5), A.18.2(79.6/5), A.18.2(97.4/5)
Has_Same_Storage attribute 13.3(73.2/4)
Hash
child of Ada.Strings A.4.9(2/5)
child of Ada.Strings.Bounded A.4.9(7/5)
child of Ada.Strings.Unbounded A.4.9(10/5)
Hash_Case_Insensitive
child of Ada.Strings A.4.9(11.2/5)
child of Ada.Strings.Bounded A.4.9(11.7/5)
child of Ada.Strings.Fixed A.4.9(11.5/3)
child of Ada.Strings.Unbounded A.4.9(11.10/5)
Hash_Type
in Ada.Containers A.18.1(4/2)
Hashed_Maps
child of Ada.Containers A.18.5(2/5)
Hashed_Sets
child of Ada.Containers A.18.8(2/5)
Head
in Ada.Strings.Bounded A.4.4(70), A.4.4(71)
in Ada.Strings.Fixed A.4.3(35), A.4.3(36)
in Ada.Strings.Unbounded A.4.5(65), A.4.5(66)
head (of a queue) D.2.1(5/2)
heap management
user-defined 13.11(1)
See also allocator 4.8(1)
held priority D.11(4/2)
heterogeneous input-output A.12.1(1)
hexadecimal
literal 2.4.2(1)
hexadecimal digit
a category of Character A.3.2(30)
hexadecimal literal 2.4.2(1)
Hexadecimal_Digit_Set
in Ada.Strings.Maps.Constants A.4.6(4)
hidden from all visibility 8.3(5), 8.3(14)
by lack of a with_clause 8.3(20/2)
for a declaration completed by a subsequent declaration 8.3(19)
for overridden declaration 8.3(15)
within the declaration itself 8.3(16)
hidden from direct visibility 8.3(5), 8.3(21)
by an inner homograph 8.3(22)
where hidden from all visibility 8.3(23)
hiding 8.3(5)
Hierarchical_File_Names
child of Ada.Directories A.16.1(3/5)
child of Ada.Wide_Directories A.16.2(2/5)
child of Ada.Wide_Wide_Directories A.16.2(3/5)
High_Order_First 13.5.3(2)
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
highest precedence operator 4.5.6(1)
highest_precedence_operator 4.5(7)
Hold
in Ada.Asynchronous_Task_Control D.11(3/5)
Holder
in Ada.Containers.Indefinite_Holders A.18.18(6/5)
homograph 8.3(8)
Hour
in Ada.Calendar.Formatting 9.6.1(24/2)
Hour_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
HT
in Ada.Characters.Latin_1 A.3.3(5)
HTJ
in Ada.Characters.Latin_1 A.3.3(17)
HTS
in Ada.Characters.Latin_1 A.3.3(17)
Hyphen
in Ada.Characters.Latin_1 A.3.3(8)
hyphen-minus 2.1(15/6)

I

i
in Ada.Numerics.Generic_Complex_Types G.1.1(5)
in Interfaces.Fortran B.5(10)
identifier 2.3(2/2)
used 2.8(2), 2.8(3/3), 2.8(21), 2.8(23), 3.1(4), 3.8(3/5), 4.1(3), 4.1.3(3), 4.1.4(3/2), 4.5.10(4/5), 5.5(2), 5.6(2), 6.1(5), 7.1(3/3), 7.2(2/3), 9.1(4), 9.1(6/3), 9.4(4), 9.4(7/3), 9.5.2(3), 9.5.2(5/5), 9.10.1(3/5), 11.4.2(6.1/3), 11.4.2(6/2), 11.5(4.1/2), 11.5(4/2), 13.1.1(3/3), 13.1.1(4/5), 13.12(4/2), 13.12(11/3), D.2.2(2), D.2.2(2.2/2), D.3(3), D.3(4), D.4(3), D.4(4/5), D.4.1(3/5), H.6(3/2), J.10(3/2), J.15.5(2/3), J.15.5(3/3), J.15.5(4/3), L(1.1/5), L(2.3/2), L(2.4/3), L(6.2/5), L(8.1/3), L(13.1/3), L(14.1/3), L(20), L(21), L(23), L(25.1/2), L(27.2/2), L(27.3/3), L(29), L(36), L(37), L(37.3/2), M.2(98), P.1
identifier specific to a pragma 2.8(10/3)
identifier_extend 2.3(3.1/3)
used 2.3(2/2), P.1
identifier_start 2.3(3/2)
used 2.3(2/2), P.1
Identity
in Ada.Strings.Maps A.4.2(22)
in Ada.Strings.Wide_Maps A.4.7(22)
in Ada.Strings.Wide_Wide_Maps A.4.8(22/2)
Identity attribute 11.4.1(9), C.7.1(12)
idle task D.11(4/2)
if expression 4.5.7(1/3)
if_expression 4.5.7(3/3)
used 4.5.7(2/3), P.1
if_statement 5.3(2)
used 5.1(5/5), P.1
illegal
construct 1.1.2(27)
partition 1.1.2(29)
Im
in Ada.Numerics.Generic_Complex_Arrays G.3.2(7/2), G.3.2(27/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(6)
image
of a value 4.10(1/5)
in Ada.Calendar.Formatting 9.6.1(35/2), 9.6.1(37/2)
in Ada.Numerics.Discrete_Random A.5.2(26)
in Ada.Numerics.Float_Random A.5.2(14)
in Ada.Task_Identification C.7.1(3/5)
in Ada.Text_IO.Editing F.3.3(13)
Image attribute 4.10(34/5), 4.10(40/5)

Imaginary
in Ada.Numerics.Generic_Complex_-Types G.1.1(4/5)
Imaginary *subtype of* Imaginary
in Interfaces.Fortran B.5(10)
immediate scope
of (a view of) an entity 8.2(11)
of a declaration 8.2(2)
of a pragma 8.2(12/4)
Immediate_Reclamation restriction H.4(10)
immediately enclosing 8.1(13/6)
immediately visible 8.3(4), 8.3(21)
immediately within 8.1(13/6)
immutably limited 7.5(8.1/3)
implementation advice 1.1.2(37)
summary of advice M.3(1/5)
implementation defined 1.1.3(18)
summary of characteristics M.2(1/5)
implementation permissions 1.1.2(36)
implementation requirements 1.1.2(33)
implementation-dependent
See unspecified 1.1.3(18)
implemented
by a protected entry 9.4(11.1/3)
by a protected subprogram 9.4(11.1/3)
by a task entry 9.1(9.2/3)
implicit conversion
legality 8.6(27.1/4)
implicit declaration 3.1(5)
implicit initial values
for a subtype 3.3.1(10)
implicit subtype conversion 4.6(59), 4.6(60)
Access attribute 3.10.2(30/5)
access discriminant 3.7(27/2)
array bounds 4.6(38)
array index 4.1.1(7)
assignment to view conversion 4.6(55/5)
assignment_statement 5.2(11/5)
bounds of a decimal fixed point type 3.5.9(16)
bounds of a fixed point type 3.5.9(14)
bounds of a range 3.5(9), 3.6(18)
choices of aggregate 4.3.3(22)
component defaults 3.3.1(13/5)
container_aggregate 4.3.5(54/6)
default value of a scalar 3.3.1(11.1/5)
delay expression 9.6(20)
derived type discriminants 3.4(21)
discriminant values 3.7.1(12/5)
entry index 9.5.2(24/5)
expressions in aggregate 4.3.1(19/5)
expressions of aggregate 4.3.3(23)
function return 6.5(5.12/5), 6.5(6/2)
generic formal object of mode in 12.4(11)
inherited enumeration literal 3.4(29)
initialization expression 3.3.1(17/5)
initialization expression of allocator 4.8(7/2)
Interrupt_Priority aspect D.1(17/4), D.3(6.1/3)
named number value 3.3.2(6)
operand of concatenation 4.5.3(9)
parameter passing 6.4.1(10), 6.4.1(11), 6.4.1(17)
Priority aspect D.1(17/4), D.3(6.1/3)
qualified_expression 4.7(4/5)
reading a view conversion 4.6(56/5)
result of inherited function 3.4(27/6)
implicit_dereference 4.1(6)
used 4.1(4), P.1
Implicit_Dereference aspect 4.1.5(2/3)
implicitly composed
aspect 13.1.1(18.3/5)
Import aspect B.1(1/3)
Import pragma J.15.5(2/3), L(14.1/3)
imported entity B.1(23/3)
in (membership test) 4.5(1), 4.5.2(2/3)
In_Range
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(10/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(12/5)
inaccessible partition E.1(7)
inactive
a task state 9(10/5)
Include
in Ada.Containers.Hashed_Maps A.18.5(22/5)
in Ada.Containers.Hashed_Sets A.18.8(21/5)
in Ada.Containers.Ordered_Maps A.18.6(21/5)
in Ada.Containers.Ordered_Sets A.18.9(20/5)
included
one range in another 3.5(4)
incomplete type 1.3.1(35/5), 3.2(4.1/2), 3.10.1(2.1/4)
incomplete view 1.3.1(37/5), 3.10.1(2.1/4)
tagged 3.10.1(2.1/4)
incomplete_type_declaration 3.10.1(2/2)
used 3.2.1(2), P.1
Increase_Indent
in Ada.Strings.Text_Buffers A.4.12(15/5)
Increment
in Interfaces.C.Pointers B.3.2(11/3)
indefinite subtype 3.3(23/5), 3.7(26)
Indefinite_Doubly_Linked_Lists
child of Ada.Containers A.18.12(2/3)
Indefinite_Hashed_Maps
child of Ada.Containers A.18.13(2/3)
Indefinite_Hashed_Sets
child of Ada.Containers A.18.15(2/3)
Indefinite_Holders
child of Ada.Containers A.18.18(5/5)
Indefinite_Multiway_Trees
child of Ada.Containers A.18.17(2/3)
Indefinite_Ordered_Maps
child of Ada.Containers A.18.14(2/3)
Indefinite_Ordered_Sets
child of Ada.Containers A.18.16(2/3)
Indefinite_Vectors
child of Ada.Containers A.18.11(2/3)
Independent aspect C.6(6.3/3)
Independent pragma J.15.8(4/3), L(14.2/3)
independent subprogram 11.6(6/5)
Independent_Components aspect C.6(6.10/3)
Independent_Components pragma J.15.8(7/3), L(14.3/3)
independently addressable 9.10(1/5)
specified C.6(8.1/4)
index
of an element of an open direct file A.8(3)
in Ada.Direct_IO A.8.4(15)
in Ada.Streams.Stream_IO A.12.1(23)
in Ada.Strings.Bounded A.4.4(43.1/2), A.4.4(43.2/6), A.4.4(44), A.4.4(45.1/2), A.4.4(45/6), A.4.4(46)
in Ada.Strings.Fixed A.4.3(8.1/2), A.4.3(8.2/6), A.4.3(9), A.4.3(10.1/2), A.4.3(10/6), A.4.3(11)
in Ada.Strings.Unbounded A.4.5(38.1/2), A.4.5(38.2/6), A.4.5(39), A.4.5(40.1/2), A.4.5(40/6), A.4.5(41)
Index attribute 6.1.1(30.2/5)
index parameter
of an array aggregate 4.3.3(6.1/5)
index range 3.6(13)
index subtype 3.6(9)
index type 3.6(9)
container aggregate 4.3.5(25/5)
indexed aggregate 4.3.5(25/5)
Index_Check 11.5(14)
[*partial*] 4.1.1(7), 4.1.2(7), 4.3.3(20.5/6), 4.3.3(29/3), 4.3.3(30), 4.3.4(20/5), 4.5.3(8/6), 4.6(51/5), 4.7(4/5), 4.8(10/2)
index_constraint 3.6.1(2)
used 3.2.2(7), P.1
Index_Error
in Ada.Strings A.4.1(5)
Index_Non_Blank
in Ada.Strings.Bounded A.4.4(46.1/2), A.4.4(47)
in Ada.Strings.Fixed A.4.3(11.1/2), A.4.3(12)

in Ada.Strings.Unbounded A.4.5(41.1/2), A.4.5(42)
index_subtype_definition 3.6(4)
used 3.6(3), P.1
indexable container object 4.1.6(5/3)
indexable container type 1.3.1(39/5), 4.1.6(5/3)
indexed aggregate 4.3.5(25/5)
indexed_component 4.1.1(2)
used 4.1(2/5), P.1
indexing
constant 4.1.6(12/3)
variable 4.1.6(16/3)
individual membership test 4.5.2(26.1/3)
indivisible C.6(10/4)
inferable discriminants B.3.3(20/6)
Information
child of Ada.Directories A.16(124/2)
child of Ada.Wide_Directories A.16.2(2/5)
child of Ada.Wide_Wide_Directories A.16.2(3/5)
information hiding
See package 7(1)
See private types and private extensions 7.3(1)
information systems C(1), F(1)
informative 1.1.2(18)
inherently mutable object 3.3(13/3)
inheritance
See derived types and classes 3.4(1/2)
See also tagged types and type extension 3.9(1)
inherited
from an ancestor type 3.4.1(11)
inherited component 3.4(11), 3.4(12)
inherited discriminant 3.4(11)
inherited entry 3.4(12)
inherited protected subprogram 3.4(12)
inherited subprogram 3.4(17/2)
initial value
reduction attribute 4.5.10(24/6)
Initial_Directory
in Ada.Directories.Hierarchical_File_Names A.16.1(12/3)
initialization
of a protected object 9.4(14)
of a protected object C.3.1(10/3), C.3.1(11/3)
of a task object 9.1(12/1), J.7.1(7)
of an object 3.3.1(18/2)
initialization expression 3.3.1(1/3), 3.3.1(4)
Initialize 7.6(2)
in Ada.Finalization 7.6(6/2), 7.6(8/2)
initialized allocator 4.8(4)
initialized by default 3.3.1(18/2)
Inline aspect 6.3.2(5.1/3)
Inline pragma J.15.1(2/3), L(15.1/3)
innermost dynamically enclosing 11.4(2)
input A.6(1/5)
Input aspect 13.13.2(38/5)
Input attribute 13.13.2(22), 13.13.2(32)
Input clause 13.3(7/2), 13.13.2(38/5)
Input'Class aspect 13.13.2(38/5)
input-output
unspecified for access types A.7(6/6)
Insert
in Ada.Containers.Doubly_Linked_Lists A.18.3(19/5), A.18.3(20/5), A.18.3(21/5)
in Ada.Containers.Hashed_Maps A.18.5(19/5), A.18.5(20/5), A.18.5(21/5)
in Ada.Containers.Hashed_Sets A.18.8(19/5), A.18.8(20/5)
in Ada.Containers.Ordered_Maps A.18.6(18/5), A.18.6(19/5), A.18.6(20/5)
in Ada.Containers.Ordered_Sets A.18.9(18/5), A.18.9(19/5)
in Ada.Containers.Vectors A.18.2(39/5), A.18.2(40/5), A.18.2(41/5), A.18.2(42/5), A.18.2(43/5)
in Ada.Strings.Bounded A.4.4(60), A.4.4(61)
in Ada.Strings.Fixed A.4.3(25), A.4.3(26)
in Ada.Strings.Unbounded A.4.5(55), A.4.5(56)
Insert_Child
in Ada.Containers.Multiway_Trees A.18.10(48/5), A.18.10(49/5), A.18.10(50/5)
Insert_Space
in Ada.Containers.Vectors A.18.2(48/5), A.18.2(49/5)
Insert_Vector
in Ada.Containers.Vectors A.18.2(36/5), A.18.2(37/5), A.18.2(38/5)
inspectable object H.3.2(5/2)
inspection point H.3.2(5/2)
Inspection_Point pragma H.3.2(3), L(16)
instance
of a generic function 12.3(13)
of a generic package 12.3(13)
of a generic procedure 12.3(13)
of a generic subprogram 12.3(13)
of a generic unit 12.3(1)
instructions for comment submission 0.2(57.22/5)
int
in Interfaces.C B.3(7)
Integer 3.5.4(11), 3.5.4(21)
in Standard A.1(12)
integer literal 2.4(1)
integer literals 3.5.4(14), 3.5.4(30)
integer type 1.3.1(40/5), 3.5.4(1)
Integer_Address
in System.Storage_Elements 13.7.1(10/3)
Integer_Arithmetic
child of System.Atomic_Operations C.6.4(3/5)
Integer_IO
in Ada.Text_IO A.10.1(52)
Integer_Literal aspect 4.2.1(3/5)
Integer_N B.2(8)
Integer_Text_IO
child of Ada A.10.8(21)
integer_type_definition 3.5.4(2)
used 3.2.1(4/2), P.1
Integer_Wide_Text_IO
child of Ada A.11(2/2)
Integer_Wide_Wide_Text_IO
child of Ada A.11(3/2)
interaction
between tasks 9(1/5)
interface 3.9.4(4/2)
limited 3.9.4(5/2)
nonlimited 3.9.4(5/2)
protected 3.9.4(5/2)
synchronized 3.9.4(5/2)
task 3.9.4(5/2)
type 3.9.4(4/2)
interface to assembly language C.1(4/3)
interface to C B.3(1/4)
interface to COBOL B.4(1/3)
interface to Fortran B.5(1/3)
interface to other languages B(1)
interface type 1.3.1(42/5)
Interface_Ancestor_Tags
in Ada.Tags 3.9(7.4/2)
interface_list 3.9.4(3/2)
used 3.4(2/2), 3.9.4(2/2), 7.3(3/3), 9.1(2/3), 9.1(3/5), 9.4(2/3), 9.4(3/3), 12.5.1(3/5), P.1
interface_type_definition 3.9.4(2/2)
used 3.2.1(4/2), 12.5.5(2/2), P.1
Interfaces B.2(3/5)
Interfaces.C B.3(4/5)
Interfaces.C.Pointers B.3.2(4/5)
Interfaces.C.Strings B.3.1(3/5)
Interfaces.COBOL B.4(7/5)
Interfaces.Fortran B.5(4/5)
Interfaces_Assertion_Check 11.5(23.7/5)
interfacing aspect B.1(0.1/3)
interfacing pragma J.15.5(1/3)
Convention J.15.5(1/3)
Export J.15.5(1/3)
Import J.15.5(1/3)
internal call 9.5(3/3)
internal code 13.4(7)
internal node
of a tree A.18.10(2/4)
internal requeue 9.5(7)

Internal_Tag
in Ada.Tags 3.9(7/2)
interpretation
of a complete context 8.6(10)
of a constituent of a complete context 8.6(15)
overload resolution 8.6(14)
interrupt C.3(2)
example using asynchronous_select 9.7.4(10), 9.7.4(12)
interrupt entry J.7.1(5)
interrupt handler C.3(2)
Interrupt_Clocks_Supported
in Ada.Execution_Time D.14(9.1/3)
Interrupt_Handler aspect C.3.1(6.2/3)
Interrupt_Handler pragma J.15.7(2/3), L(17.1/3)
Interrupt_Id
in Ada.Interrupts C.3.2(2/5)
Interrupt_Priority aspect D.1(6.3/3)
Interrupt_Priority pragma J.15.11(4/3), L(18.1/3)
Interrupt_Priority *subtype of* Any_Priority
in System 13.7(16)
Interrupts
child of Ada C.3.2(2/5)
child of Ada.Execution_Time D.14.3(3/5)
Intersection
in Ada.Containers.Hashed_Sets A.18.8(29/5), A.18.8(30/5)
in Ada.Containers.Ordered_Sets A.18.9(30/5), A.18.9(31/5)
intertask communication 9.5(1)
See also task 9(1/5)
Intrinsic calling convention 6.3.1(4)
invalid cursor
of a list container A.18.3(153/2)
of a map A.18.4(76/2)
of a set A.18.7(97/2)
of a tree A.18.10(222/3)
of a vector A.18.2(248/2)
invalid representation 13.9.1(9)
invariant 1.3.1(44/5), 7.3.2(2/3)
class-wide 7.3.2(3/5)
invariant check 7.3.2(9/4)
invariant expression 7.3.2(2/3)
Inverse
in Ada.Numerics.Generic_Complex_-Arrays G.3.2(46/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Inverted_Exclamation
in Ada.Characters.Latin_1 A.3.3(21/3)
Inverted_Question
in Ada.Characters.Latin_1 A.3.3(22)
involve an inner product
complex G.3.2(56/2)
real G.3.1(34/2)
IO_Assertion_Check 11.5(23.8/5)
IO_Exceptions
child of Ada A.13(3/5)
IS1
in Ada.Characters.Latin_1 A.3.3(16)
IS2
in Ada.Characters.Latin_1 A.3.3(16)
IS3
in Ada.Characters.Latin_1 A.3.3(16)
IS4
in Ada.Characters.Latin_1 A.3.3(16)
Is_A_Group_Member
in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_Abstract
in Ada.Tags 3.9(7.5/3)
Is_Alphanumeric
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(12/3)
Is_Ancestor_Of
in Ada.Containers.Multiway_Trees A.18.10(21.2/5)
Is_Attached
in Ada.Interrupts C.3.2(5)
Is_Basic
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(8.1/5)
Is_Callable
in Ada.Task_Identification C.7.1(4/3)
Is_Character
in Ada.Characters.Conversions A.3.4(3/2)
Is_Control
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(5/3)
Is_Current_Directory_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(7/3)
Is_Decimal_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(10/3)
Is_Descendant_At_Same_Level
in Ada.Tags 3.9(7.1/2)
Is_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(9/3)
Is_Empty
in Ada.Containers.Doubly_Linked_-Lists A.18.3(12/5)
in Ada.Containers.Hashed_Maps A.18.5(11/5)
in Ada.Containers.Hashed_Sets A.18.8(13/5)
in Ada.Containers.Indefinite_Holders A.18.18(10/5)
in Ada.Containers.Multiway_Trees A.18.10(16/5)
in Ada.Containers.Ordered_Maps A.18.6(10/5)
in Ada.Containers.Ordered_Sets A.18.9(12/5)
in Ada.Containers.Vectors A.18.2(23/5)
Is_Full_Name
in Ada.Directories.Hierarchical_File_Names A.16.1(8/3)
Is_Graphic
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(19/3)
Is_Held
in Ada.Asynchronous_Task_Control D.11(3/5)
Is_Hexadecimal_Digit
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(11/3)
Is_In
in Ada.Strings.Maps A.4.2(13)
in Ada.Strings.Wide_Maps A.4.7(13)
in Ada.Strings.Wide_Wide_Maps A.4.8(13/2)
Is_ISO_646
in Ada.Characters.Handling A.3.2(10)
Is_Leaf
in Ada.Containers.Multiway_Trees A.18.10(21.1/5), A.18.10(21/5)
Is_Letter
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(6/3)
Is_Line_Terminator
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(14/3)
Is_Lock_Free
in System.Atomic_Operations.Exchange C.6.2(8/5)
in System.Atomic_Operations.Integer_Arithmetic C.6.4(8/5)
in System.Atomic_Operations.Modular_Arithmetic C.6.5(8/5)
in System.Atomic_Operations.Test_And_Set C.6.3(7/5)
Is_Lower
in Ada.Characters.Handling A.3.2(4/5)
in Ada.Wide_Characters.Handling A.3.5(7/3)

Is_Mark
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(15/3)
Is_Member
  in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Is_NFKC
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(18.1/5)
Is_Nul_Terminated
  in Interfaces.C B.3(24), B.3(35), B.3(39.16/2), B.3(39.7/2)
Is_Open
  in Ada.Direct_IO A.8.4(10)
  in Ada.Sequential_IO A.8.1(10)
  in Ada.Streams.Stream_IO A.12.1(12)
  in Ada.Text_IO A.10.1(13)
Is_Other_Format
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(16/3)
Is_Parent_Directory_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(6/3)
Is_Punctuation_Connector
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(17/3)
Is_Relative_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(9/3)
Is_Reserved
  in Ada.Interrupts C.3.2(4)
Is_Root
  in Ada.Containers.Multiway_Trees A.18.10(20.1/5), A.18.10(20/5)
Is_Root_Directory_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(5/3)
Is_Round_Robin
  in Ada.Dispatching.Round_Robin D.2.5(4/5)
Is_Simple_Name
  in Ada.Directories.Hierarchical_File_Names A.16.1(4/3)
Is_Sorted
  in Ada.Containers.Doubly_Linked_Lists A.18.3(48/2)
  in Ada.Containers.Vectors A.18.2(76/2)
Is_Space
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(18/3)
Is_Special
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(13/3)
Is_Split
  in Ada.Iterator_Interfaces 5.5.1(4.3/5)
Is_String
  in Ada.Characters.Conversions A.3.4(3/2)
Is_Subset
  in Ada.Containers.Hashed_Sets A.18.8(39/2)
  in Ada.Containers.Ordered_Sets A.18.9(40/2)
  in Ada.Strings.Maps A.4.2(14)
  in Ada.Strings.Wide_Maps A.4.7(14)
  in Ada.Strings.Wide_Wide_Maps A.4.8(14/2)
Is_Terminated
  in Ada.Task_Identification C.7.1(4/3)
Is_Upper
  in Ada.Characters.Handling A.3.2(4/5)
  in Ada.Wide_Characters.Handling A.3.5(8/3)
Is_Valid
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(4/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(4/5)
Is_Wide_Character
  in Ada.Characters.Conversions A.3.4(3/2)
Is_Wide_String
  in Ada.Characters.Conversions A.3.4(3/2)
ISO 1989:2002 1.2.1(4/5)
ISO 3166-1:2020 1.2(4.1/5)
ISO 639-3:2007 1.2(1.1/3)
ISO 8601-1:2019 1.2.1(6/5)
ISO/IEC 10646:2020 1.2(8/6), 1.2.1(8.1/6), 3.5.2(2/6), 3.5.2(3/6), 3.5.2(4/6)
ISO/IEC 14882:2020 1.2.1(9/5)
ISO/IEC 1539-1:2018 1.2.1(3/5)
ISO/IEC 60559:2020 1.2.1(10/5)
ISO/IEC 6429:1992 1.2.1(5/5)
ISO/IEC 646:1991 1.2.1(2/5)
ISO/IEC 8859-1:1998 1.2.1(7/5)
ISO/IEC 9899:2018 1.2.1(8/5)
ISO_646 *subtype of* Character
  in Ada.Characters.Handling A.3.2(9)
ISO_646_Set
  in Ada.Strings.Maps.Constants A.4.6(4)
issue
  an entry call 9.5.3(8)
italics
  nongraphic characters 3.5.2(2/6)
  pseudo-names of anonymous types 3.2.1(7/2), A.1(2)
  syntax rules 1.1.4(14)
  terms introduced or defined 1.3(1/5)
iterable container object 5.5.1(11/5)
iterable container object for a loop 5.5.2(12/6)
iterable container type 1.3.1(45/5), 5.5.1(11/5)
Iterate
  in Ada.Containers.Doubly_Linked_Lists A.18.3(45/5), A.18.3(46.1/5), A.18.3(46.2/5)
  in Ada.Containers.Hashed_Maps A.18.5(37/5)
  in Ada.Containers.Hashed_Sets A.18.8(49/5)
  in Ada.Containers.Multiway_Trees A.18.10(42/5), A.18.10(44/5)
  in Ada.Containers.Ordered_Maps A.18.6(50/5), A.18.6(51.1/5), A.18.6(51.2/5)
  in Ada.Containers.Ordered_Sets A.18.9(60/5)
  in Ada.Containers.Vectors A.18.2(73/5), A.18.2(74.1/5), A.18.2(74.2/5)
  in Ada.Environment_Variables A.17(8/5)
Iterate_Children
  in Ada.Containers.Multiway_Trees A.18.10(68.1/5), A.18.10(68/5), A.18.10(70/5)
Iterate_Subtree
  in Ada.Containers.Multiway_Trees A.18.10(43.1/5), A.18.10(43/5), A.18.10(45.1/5), A.18.10(45/5)
iterated_component_association 4.3.3(5.1/5)
  used 4.3.3(5/5), P.1
iterated_element_association 4.3.5(21/5)
  used 4.3.5(18/5), 4.5.10(3/5), P.1
iterating procedure 5.5.3(9/5), 5.5.3(27/5)
iteration cursor subtype 5.5.1(6/5)
iteration_scheme 5.5(3/5)
  used 5.5(2), P.1
iterative context 4.4(9.8/6)
iterator 1.3.3(16/5)
  array component 5.5.2(3/3)
  container element 5.5.2(3/3)
  forward 5.5.2(4/5)
  generalized 5.5.2(3/3)
  parallel 5.5.2(4/5)
  reverse 5.5.2(4/5)
  sequential 5.5.2(4/5)

iterator construct 5.5(6.2/5)
iterator filter 1.3.3(18/5), 5.5(6.2/5)
iterator object 5.5.1(6/5)
iterator type 5.5.1(6/5)
iterator_actual_parameter_part 5.5.3(5/5)
used 5.5.3(4/5), P.1
Iterator_Element aspect 5.5.1(9/5)
iterator_filter 5.5(4.1/5)
used 5.5(4/5), 5.5.2(2/5), 5.5.3(2/5), P.1
Iterator_Interfaces
child of Ada 5.5.1(2/5)
iterator_parameter_association 5.5.3(6/5)
used 5.5.3(5/5), P.1
iterator_parameter_specification 5.5.3(3/5)
used 5.5.3(2/5), P.1
iterator_procedure_call 5.5.3(4/5)
used 5.5.3(2/5), P.1
iterator_specification 5.5.2(2/5)
used 4.3.3(5.1/5), 4.3.5(21/5), 4.5.8(1/3), 5.5(3/5), P.1
Iterator_View aspect 5.5.1(9.1/5)

J

j
in Ada.Numerics.Generic_Complex_-Types G.1.1(5)
in Interfaces.Fortran B.5(10)
Jorvik profile D.13(6.1/5)

K

Key
in Ada.Containers.Hashed_Maps A.18.5(13.1/5), A.18.5(13/5)
in Ada.Containers.Hashed_Sets A.18.8(51.1/5), A.18.8(51/5)
in Ada.Containers.Ordered_Maps A.18.6(12.1/5), A.18.6(12/5)
in Ada.Containers.Ordered_Sets A.18.9(64.1/5), A.18.9(64/5)
key type
container aggregate 4.3.5(8/6), 4.3.5(9/6)
key_choice 4.3.5(20/5)
used 4.3.5(19/5), P.1
key_choice_list 4.3.5(19/5)
used 4.3.5(18/5), P.1
Kind
in Ada.Directories A.16(25/2), A.16(40/2)
known discriminants 3.7(26)
known on entry
postcondition 6.1.1(20/5)
known to be constrained 3.3(23.1/3)
known to denote the same object 6.4.1(6.5/3)
known to have no variable views 3.3(13.1/5)
known to refer to the same object 6.4.1(6.12/3)
Known_Conflict_Checks conflict check policy 9.10.1(22/5)
known_discriminant_part 3.7(4)
used 3.2.1(3/3), 3.7(2/2), 9.1(2/3), 9.4(2/3), P.1
Known_Parallel_Conflict_Checks conflict check policy 9.10.1(12/5)
Known_Tasking_Conflict_Checks conflict check policy 9.10.1(16/5)

L

label 5.1(7)
used 5.1(2/3), 5.1(3), P.1
Landau symbol O(X) A.18(3/2)
language
interface to assembly C.1(4/3)
interface to non-Ada B(1)
in Ada.Locales A.19(6/3)
Language code standard 1.2(1.1/3)
language-defined categories [*partial*] 3.2(10/2)
language-defined category
of types 3.2(2/5)
language-defined check 11.5(2/3), 11.6(1/3)
language-defined class [*partial*] 3.2(10/2)
of types 3.2(2/5)
Language-defined constants Q.5(1/3)
Language-defined exceptions Q.4(1/3)
Language-Defined Library Units A(1)
Language-defined objects Q.5(1/3)
Language-defined packages Q.1(1/3)
Language-defined subprograms Q.3(1/3)
Language-defined subtypes Q.2(1/3)
Language-defined types Q.2(1/3)
Language-defined values Q.5(1/3)
Language_Code
in Ada.Locales A.19(4/4)
Language_Unknown
in Ada.Locales A.19(5/3)
Last
in Ada.Containers.Doubly_Linked_-Lists A.18.3(35/5)
in Ada.Containers.Ordered_Maps A.18.6(31/5)
in Ada.Containers.Ordered_Sets A.18.9(43/5)
in Ada.Containers.Vectors A.18.2(61/5)
in Ada.Iterator_Interfaces 5.5.1(4/3)
Last attribute 3.5(13), 3.6.2(5)
last element
of a hashed set A.18.8(68/2)
of a set A.18.7(6/2)
of an ordered set A.18.9(81/3)
last node
of a hashed map A.18.5(46/2)
of a map A.18.4(6/2)
of an ordered map A.18.6(58/3)
Last(N) attribute 3.6.2(6)
last_bit 13.5.1(6)
used 13.5.1(3), P.1
Last_Bit attribute 13.5.2(4/2)
Last_Child
in Ada.Containers.Multiway_Trees A.18.10(62.1/5), A.18.10(62/5)
Last_Child_Element
in Ada.Containers.Multiway_Trees A.18.10(63.1/5), A.18.10(63/5)
Last_Element
in Ada.Containers.Doubly_Linked_-Lists A.18.3(36/5)
in Ada.Containers.Ordered_Maps A.18.6(32/5)
in Ada.Containers.Ordered_Sets A.18.9(44/5)
in Ada.Containers.Vectors A.18.2(62/5)
Last_Index
in Ada.Containers.Vectors A.18.2(60/5)
Last_Key
in Ada.Containers.Ordered_Maps A.18.6(33/5)
Last_Valid attribute 3.5.5(7.3/4)
lateness D.9(12/5)
Latin-1 3.5.2(2/6)
Latin_1
child of Ada.Characters A.3.3(3/5)
Layout aspect 13.5(1)
Layout_Error
in Ada.IO_Exceptions A.13(4)
in Ada.Text_IO A.10.1(85/5)
LC_A
in Ada.Characters.Latin_1 A.3.3(13)
LC_A_Acute
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Grave
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Ring
in Ada.Characters.Latin_1 A.3.3(25)
LC_A_Tilde
in Ada.Characters.Latin_1 A.3.3(25)
LC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(25)
LC_B
in Ada.Characters.Latin_1 A.3.3(13)
LC_C
in Ada.Characters.Latin_1 A.3.3(13)

LC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(25)
LC_D
in Ada.Characters.Latin_1 A.3.3(13)
LC_E
in Ada.Characters.Latin_1 A.3.3(13)
LC_E_Acute
in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_E_Grave
in Ada.Characters.Latin_1 A.3.3(25)
LC_F
in Ada.Characters.Latin_1 A.3.3(13)
LC_G
in Ada.Characters.Latin_1 A.3.3(13)
LC_German_Sharp_S
in Ada.Characters.Latin_1 A.3.3(24)
LC_H
in Ada.Characters.Latin_1 A.3.3(13)
LC_I
in Ada.Characters.Latin_1 A.3.3(13)
LC_I_Acute
in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Circumflex
in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Diaeresis
in Ada.Characters.Latin_1 A.3.3(25)
LC_I_Grave
in Ada.Characters.Latin_1 A.3.3(25)
LC_Icelandic_Eth
in Ada.Characters.Latin_1 A.3.3(26)
LC_Icelandic_Thorn
in Ada.Characters.Latin_1 A.3.3(26)
LC_J
in Ada.Characters.Latin_1 A.3.3(13)
LC_K
in Ada.Characters.Latin_1 A.3.3(13)
LC_L
in Ada.Characters.Latin_1 A.3.3(13)
LC_M
in Ada.Characters.Latin_1 A.3.3(13)
LC_N
in Ada.Characters.Latin_1 A.3.3(13)
LC_N_Tilde
in Ada.Characters.Latin_1 A.3.3(26)
LC_O
in Ada.Characters.Latin_1 A.3.3(13)
LC_O_Acute
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Circumflex
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Grave
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Oblique_Stroke
in Ada.Characters.Latin_1 A.3.3(26)
LC_O_Tilde
in Ada.Characters.Latin_1 A.3.3(26)
LC_P
in Ada.Characters.Latin_1 A.3.3(14)
LC_Q
in Ada.Characters.Latin_1 A.3.3(14)
LC_R
in Ada.Characters.Latin_1 A.3.3(14)
LC_S
in Ada.Characters.Latin_1 A.3.3(14)
LC_T
in Ada.Characters.Latin_1 A.3.3(14)
LC_U
in Ada.Characters.Latin_1 A.3.3(14)
LC_U_Acute
in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Circumflex
in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_U_Grave
in Ada.Characters.Latin_1 A.3.3(26)
LC_V
in Ada.Characters.Latin_1 A.3.3(14)
LC_W
in Ada.Characters.Latin_1 A.3.3(14)
LC_X
in Ada.Characters.Latin_1 A.3.3(14)
LC_Y
in Ada.Characters.Latin_1 A.3.3(14)
LC_Y_Acute
in Ada.Characters.Latin_1 A.3.3(26)
LC_Y_Diaeresis
in Ada.Characters.Latin_1 A.3.3(26)
LC_Z
in Ada.Characters.Latin_1 A.3.3(14)
Leading_Nonseparate
in Interfaces.COBOL B.4(23)
Leading_Part attribute A.5.3(54)
Leading_Separate
in Interfaces.COBOL B.4(23)
leaf node
of a tree A.18.10(4/3)
Leap_Seconds_Count *subtype of* Integer
in Ada.Calendar.Arithmetic 9.6.1(11/2)
leaving 7.6.1(3/6)
left 7.6.1(3/6)
left parenthesis 2.1(15/6)
left square bracket 2.1(15/6)
Left_Angle_Quotation
in Ada.Characters.Latin_1 A.3.3(21/3)
Left_Curly_Bracket
in Ada.Characters.Latin_1 A.3.3(14)
Left_Parenthesis
in Ada.Characters.Latin_1 A.3.3(8)
Left_Square_Bracket
in Ada.Characters.Latin_1 A.3.3(12)
legal
construct 1.1.2(27)
partition 1.1.2(29)
legality rules 1.1.2(27)
length
of a dimension of an array 3.6(13)
of a list container A.18.3(3/2)
of a map A.18.4(5/2)
of a one-dimensional array 3.6(13)
of a set A.18.7(5/2)
of a vector container A.18.2(2/2)
in Ada.Containers.Doubly_Linked_-Lists A.18.3(11/5)
in Ada.Containers.Hashed_Maps A.18.5(10/5)
in Ada.Containers.Hashed_Sets A.18.8(12/5)
in Ada.Containers.Ordered_Maps A.18.6(9/5)
in Ada.Containers.Ordered_Sets A.18.9(11/5)
in Ada.Containers.Vectors A.18.2(21/5)
in Ada.Strings.Bounded A.4.4(9)
in Ada.Strings.Unbounded A.4.5(6)
in Ada.Text_IO.Editing F.3.3(11)
in Interfaces.COBOL B.4(34), B.4(39), B.4(44)
Length attribute 3.6.2(9)
Length(N) attribute 3.6.2(10)
Length_Check 11.5(15)
[*partial*] 4.5.1(8), 4.6(37), 4.6(52)
Length_Error
in Ada.Strings A.4.1(5)
Length_Range *subtype of* Natural
in Ada.Strings.Bounded A.4.4(8)
less than operator 4.5(1), 4.5.2(1)
less than or equal operator 4.5(1), 4.5.2(1)
less-than sign 2.1(15/6)
Less_Case_Insensitive
child of Ada.Strings A.4.10(13/5)
child of Ada.Strings.Bounded A.4.10(18/5)
child of Ada.Strings.Fixed A.4.10(16/3)
child of Ada.Strings.Unbounded A.4.10(21/5)
Less_Than_Sign
in Ada.Characters.Latin_1 A.3.3(10)
letter
a category of Character A.3.2(24)
letter_lowercase 2.1(9/2)
used 2.3(3/2), P.1
letter_modifier 2.1(9.2/2)
used 2.3(3/2), P.1
letter_other 2.1(9.3/2)
used 2.3(3/2), P.1
Letter_Set
in Ada.Strings.Maps.Constants A.4.6(4)
letter_titlecase 2.1(9.1/2)
used 2.3(3/2), P.1

letter_uppercase 2.1(8/2)
used 2.3(3/2), P.1
level
accessibility 3.10.2(3/5)
library 3.10.2(22)
lexical element 2.2(1)
lexicographic order 4.5.2(26/3)
LF
in Ada.Characters.Latin_1 A.3.3(5)
library 10.1.4(9/5)
[*partial*] 10.1.1(9)
informal introduction 10(2)
See also library level, library unit, library_item
library level 3.10.2(22)
library unit 1.3.3(19/5), 10.1(3), 10.1.1(9)
informal introduction 10(2)
See also language-defined library units
library unit aspect 13.1.1(32/6)
library unit pragma J.15(7/5)
All_Calls_Remote J.15.15(7/5)
Elaborate_Body J.15.14(7/5)
Preelaborate J.15.14(7/5)
Pure J.15.14(7/5)
Remote_Call_Interface J.15.15(7/5)
Remote_Types J.15.15(7/5)
Shared_Passive J.15.15(7/5)
library_item 10.1.1(4)
informal introduction 10(2)
used 10.1.1(3), P.1
library_unit_body 10.1.1(7)
used 10.1.1(4), P.1
library_unit_declaration 10.1.1(5)
used 10.1.1(4), P.1
library_unit_renaming_declaration 10.1.1(6)
used 10.1.1(4), P.1
lifetime 3.10.2(3/5)
limited interface 3.9.4(5/2)
limited type 1.3.1(46/5), 7.5(3/3)
becoming nonlimited 7.3.1(5/1), 7.5(16)
immutably 7.5(8.1/3)
limited view 10.1.1(12.1/2)
Limited_Controlled
in Ada.Finalization 7.6(7/5)
limited_with_clause 10.1.2(4.1/2)
used 10.1.2(4/2), P.1
line 2.2(2/3)
in Ada.Text_IO A.10.1(38/5)
line terminator A.10(7)
Line_Length
in Ada.Text_IO A.10.1(25/5)
link name B.1(35)
link-time error
See post-compilation error 1.1.2(29)
See post-compilation error 1.1.5(4)
Link_Name aspect B.1(1/3)
Linker_Options pragma B.1(8), L(19)
linking
See partition building 10.2(2/5)
List
in Ada.Containers.Doubly_Linked_Lists A.18.3(6/5), A.18.3(51.2/5)
list container A.18.3(1/2)
List pragma 2.8(21), L(20)
List_Iterator_Interfaces
in Ada.Containers.Doubly_Linked_Lists A.18.3(9.3/3), A.18.3(51.7/5)
literal 4.2(1)
based 2.4.2(1)
decimal 2.4.1(1)
numeric 2.4(1)
See also aggregate 4.3(1)
little endian 13.5.3(2)
load time C.4(3)
local to 8.1(14)
Local_Image
in Ada.Calendar.Formatting 9.6.1(35.1/5)
local_name 13.1(3)
used 13.3(2), 13.4(2), 13.5.1(2/5), 13.5.1(3), C.5(3), J.15.2(2/5), J.15.3(2/3), J.15.5(2/3), J.15.5(3/3), J.15.5(4/3), J.15.6(2/3), J.15.8(2/3), J.15.8(3/3), J.15.8(4/3), J.15.8(5/3), J.15.8(6/3), J.15.8(7/3), J.15.13(2/3), L(3.1/3), L(4.1/3), L(5.1/3), L(8.1/3), L(9), L(13.1/3), L(14.1/3), L(14.2/3), L(14.3/3), L(21.2/3), L(24.1/3), L(37.2/3), L(38.1/3), L(39.1/3), P.1
Local_Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(6/5)
locale A.19(1/3)
active A.19(8/3)
Locales
child of Ada A.19(3/5)
lock-free C.6.1(5/5)
locking policy D.3(6/2)
Ceiling_Locking D.3(7)
Locking_Policy pragma D.3(3), L(21)
Log
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
Logical
in Interfaces.Fortran B.5(7)
logical operator 4.5.1(2)
See also not operator 4.5.6(3)
logical thread of control 1.3.4(9/5), 9(1/5)
logical_operator 4.5(2)
long
in Interfaces.C B.3(7)
Long_Binary
in Interfaces.COBOL B.4(10)
long_double
in Interfaces.C B.3(17)
Long_Float 3.5.7(15), 3.5.7(16/5), 3.5.7(17)
Long_Floating
in Interfaces.COBOL B.4(9)
Long_Integer 3.5.4(22), 3.5.4(25), 3.5.4(28)
Look_Ahead
in Ada.Text_IO A.10.1(43/5)
loop
parallel 5.5(8.1/5)
sequential 5.5(9/5)
loop body procedure 5.5.3(15/5)
loop cursor 5.5.2(12/6)
loop iterator 5.5.2(10/6)
container element iterator 5.5.2(12/6)
loop parameter 5.5(6/5), 5.5.2(7/5)
loop_parameter_specification 5.5(4/5)
used 4.3.5(21/5), 4.5.8(1/3), 5.5(3/5), P.1
loop_parameter_subtype_indication 5.5.2(2.1/5)
used 5.5.2(2/5), P.1
loop_statement 5.5(2)
used 5.1(5/5), P.1
low line 2.1(15/6)
low-level programming C(1)
Low_Line
in Ada.Characters.Latin_1 A.3.3(12)
Low_Order_First 13.5.3(2)
in Interfaces.COBOL B.4(25)
in System 13.7(15/2)
lower bound
of a range 3.5(4)
lower-case letter
a category of Character A.3.2(25)
Lower_Case_Map
in Ada.Strings.Maps.Constants A.4.6(5)
Lower_Set
in Ada.Strings.Maps.Constants A.4.6(4)

M

Machine attribute A.5.3(60)
machine code insertion 13.8(1), C.1(2/5)
machine numbers
of a fixed point type 3.5.9(8/2)
of a floating point type 3.5.7(8)
machine scalar 13.3(8.1/3)
Machine_Code
child of System 13.8(7)
Machine_Emax attribute A.5.3(8)
Machine_Emin attribute A.5.3(7)
Machine_Mantissa attribute A.5.3(6)
Machine_Overflows attribute A.5.3(12), A.5.4(4)
Machine_Radix aspect F.1(1/5)

Machine_Radix attribute A.5.3(2), A.5.4(2)
Machine_Radix clause 13.3(7/2), F.1(1/5)
Machine_Rounding attribute A.5.3(41.1/2), A.5.4(11/6)
Machine_Rounds attribute A.5.3(11), A.5.4(3)
macro
  See generic unit 12(1)
Macron
  in Ada.Characters.Latin_1 A.3.3(21/3)
main subprogram
  for a partition 10.2(7)
malloc
  See allocator 4.8(1)
Map
  in Ada.Containers.Hashed_Maps A.18.5(3/5), A.18.5(37.3/5)
  in Ada.Containers.Ordered_Maps A.18.6(4/5), A.18.6(51.4/5)
map container A.18.4(1/2)
Map_Iterator_Interfaces
  in Ada.Containers.Hashed_Maps A.18.5(6.3/5), A.18.5(37.8/5)
  in Ada.Containers.Ordered_Maps A.18.6(7.3/5), A.18.6(51.9/5)
Maps
  child of Ada.Strings A.4.2(3/5)
mark_non_spacing 2.1(9.4/2), 2.1(9.5/2)
  used 2.3(3.1/3), P.1
mark_spacing_combining
  used 2.3(3.1/3), P.1
marshalling E.4(9)
Masculine_Ordinal_Indicator
  in Ada.Characters.Latin_1 A.3.3(22)
master 1.3.4(10/5), 7.6.1(3/6)
master construct 1.3.3(21/5), 7.6.1(3/6)
master of a call 3.10.2(10.1/3)
match
  a character to a pattern character A.4.2(54)
  a character to a pattern character, with respect to a character mapping function A.4.2(64)
  a string to a pattern string A.4.2(54)
  value of nonoverridable aspect 13.1.1(18.5/6)
matching components 4.5.2(16)
Max
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Max attribute 3.5(19)
Max_Alignment_For_Allocation attribute 13.11.1(4/5)
Max_Asynchronous_Select_Nesting restriction D.7(18/1)
Max_Base_Digits 3.5.7(6)
  in System 13.7(8)
Max_Binary_Modulus 3.5.4(7)
  in System 13.7(7)
Max_Decimal_Digits
  in Ada.Decimal F.2(5)
Max_Delta
  in Ada.Decimal F.2(4)
Max_Digits 3.5.7(6)
  in System 13.7(8)
Max_Digits_Binary
  in Interfaces.COBOL B.4(11)
Max_Digits_Long_Binary
  in Interfaces.COBOL B.4(11)
Max_Entry_Queue_Length aspect D.4(18/5)
Max_Entry_Queue_Length restriction D.7(19.1/2)
Max_Image_Length restriction H.4(23.11/5)
Max_Image_Width
  in Ada.Numerics.Discrete_Random A.5.2(25)
  in Ada.Numerics.Float_Random A.5.2(13)
Max_Int 3.5.4(14)
  in System 13.7(6)
Max_Length
  in Ada.Strings.Bounded A.4.4(5)
Max_Mantissa
  in System 13.7(9)
Max_Nonbinary_Modulus 3.5.4(7)
  in System 13.7(7)
Max_Picture_Length
  in Ada.Text_IO.Editing F.3.3(8)
Max_Protected_Entries restriction D.7(14)
Max_Scale
  in Ada.Decimal F.2(3)
Max_Select_Alternatives restriction D.7(12)
Max_Size_In_Storage_Elements attribute 13.11.1(3/5)
Max_Storage_At_Blocking restriction D.7(17/1)
Max_Task_Entries restriction D.7(13)
Max_Tasks restriction D.7(19/1)
maximum box error
  for a component of the result of evaluating a complex function G.2.6(3/5)
maximum line length A.10(11)
maximum page length A.10(11)
maximum relative error
  for a component of the result of evaluating a complex function G.2.6(3/5)
  for the evaluation of an elementary function G.2.4(2)
Maximum_Length
  in Ada.Containers.Vectors A.18.2(12.3/5)
Meaningful_For
  in Ada.Containers.Multiway_Trees A.18.10(22.1/5)
Members
  in Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Membership
  in Ada.Strings A.4.1(6)
membership test 4.5.2(2/3)
membership_choice 4.4(3.2/4)
  used 4.4(3.1/5), P.1
membership_choice_list 4.4(3.1/5)
  used 4.4(3/4), P.1
Memory_Size
  in System 13.7(13)
mentioned
  in a with_clause 10.1.2(6/2)
Merge
  in Ada.Containers.Doubly_Linked_Lists A.18.3(50/5)
  in Ada.Containers.Vectors A.18.2(78/5)
message
  See dispatching call 3.9.2(1/5)
method
  See dispatching subprogram 3.9.2(1/5)
metrics 1.1.2(35/6)
Micro_Sign
  in Ada.Characters.Latin_1 A.3.3(22)
Microseconds
  in Ada.Real_Time D.8(14/2)
Middle_Dot
  in Ada.Characters.Latin_1 A.3.3(22)
Milliseconds
  in Ada.Real_Time D.8(14/2)
Min
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(18/5)
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(21/5)
Min attribute 3.5(16)
Min_Delta
  in Ada.Decimal F.2(4)

Min_Handler_Ceiling
in Ada.Execution_Time.Group_Budgets D.14.2(7/2)
in Ada.Execution_Time.Timers D.14.1(6/2)
Min_Int 3.5.4(14)
in System 13.7(6)
Min_Scale
in Ada.Decimal F.2(3)
minus 2.1(15/6)
minus operator 4.5(1), 4.5.3(1), 4.5.4(1)
Minus_Sign
in Ada.Characters.Latin_1 A.3.3(8)
Minute
in Ada.Calendar.Formatting 9.6.1(25/2)
Minute_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
Minutes
in Ada.Real_Time D.8(14/2)
mixed-language programs B(1), C.1(4/3)
Mod attribute 3.5.4(16.1/2)
mod operator 4.5(1), 4.5.5(1)
mod_clause J.8(1)
used 13.5.1(2/5), P.1
mode 6.1(16)
used 6.1(15/5), 12.4(2/3), P.1
in Ada.Direct_IO A.8.4(9)
in Ada.Sequential_IO A.8.1(9)
in Ada.Streams.Stream_IO A.12.1(11)
in Ada.Text_IO A.10.1(12)
mode conformance 6.3.1(16/3)
required 8.5.4(4/5), 8.5.4(5/3), 12.6(7/3), 12.6(8/3), 13.1(13.2/5)
mode of operation
nonstandard 1.1.5(11)
standard 1.1.5(11)
Mode_Error
in Ada.Direct_IO A.8.4(18)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
Model attribute A.5.3(68), G.2.2(7)
model interval G.2.1(4)
associated with a value G.2.1(4)
model number G.2.1(3)
model-oriented attributes
of a floating point subtype A.5.3(63)
Model_Emin attribute A.5.3(65), G.2.2(4)
Model_Epsilon attribute A.5.3(66)
Model_Mantissa attribute A.5.3(64), G.2.2(3/2)
Model_Small attribute A.5.3(67)
Modification_Time
in Ada.Directories A.16(27/2), A.16(42/2)
modular type 3.5.4(1)
Modular_Arithmetic
child of System.Atomic_Operations C.6.5(3/5)
Modular_IO
in Ada.Text_IO A.10.1(57)
modular_type_definition 3.5.4(4)
used 3.5.4(2), P.1
module
See package 7(1)
modulus
of a modular type 3.5.4(7)
in Ada.Numerics.Generic_Complex_-Arrays G.3.2(10/2), G.3.2(30/2)
in Ada.Numerics.Generic_Complex_-Types G.1.1(9)
Modulus attribute 3.5.4(17)
Monday
in Ada.Calendar.Formatting 9.6.1(17/2)
Month
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(22/2)
Month_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)
More_Entries
in Ada.Directories A.16(34/2)
Move
in Ada.Containers.Doubly_Linked_-Lists A.18.3(18/5)
in Ada.Containers.Hashed_Maps A.18.5(18/5)
in Ada.Containers.Hashed_Sets A.18.8(18/5)
in Ada.Containers.Indefinite_Holders A.18.18(22/5)
in Ada.Containers.Multiway_Trees A.18.10(34/5)
in Ada.Containers.Ordered_Maps A.18.6(17/5)
in Ada.Containers.Ordered_Sets A.18.9(17/5)
in Ada.Containers.Vectors A.18.2(35/5)
in Ada.Strings.Fixed A.4.3(7)
multidimensional array 3.6(12)
Multiplication_Sign
in Ada.Characters.Latin_1 A.3.3(24)
multiply 2.1(15/6)
multiply operator 4.5(1), 4.5.5(1)
multiplying operator 4.5.5(1)
multiplying_operator 4.5(6)
used 4.4(5), P.1
Multiprocessors
child of System D.16(3/5)
Multiway_Trees
child of Ada.Containers A.18.10(7/5)
mutates 7.6(17.6/3)
MW
in Ada.Characters.Latin_1 A.3.3(18)

N

n-dimensional array_aggregate 4.3.3(6)
NAK
in Ada.Characters.Latin_1 A.3.3(6)
name 4.1(2/5)
[*partial*] 3.1(1/5)
of (a view of) an entity 3.1(8)
of a pragma 2.8(9)
of an external file A.7(1)
used 2.8(3/3), 3.2.2(4), 4.1(4), 4.1(5), 4.1(6), 4.1.5(4/3), 4.4(7/5), 4.5.10(5/6), 4.6(2), 4.8(2.1/3), 5.2(2), 5.5.2(2/5), 5.5.3(4/5), 5.7(2), 5.8(2), 6.1.2(8/5), 6.4(2), 6.4(3), 6.4(6), 8.4(3), 8.5.1(2/5), 8.5.2(2/3), 8.5.3(2/3), 8.5.4(2/3), 8.5.5(2/3), 9.5.3(2), 9.5.4(2/3), 9.8(2), 10.1.1(8), 10.1.2(4.1/2), 10.1.2(4.2/2), 10.2.1(20), 10.2.1(21), 11.2(5), 11.3(2.1/4), 11.3(2/2), 12.3(2/3), 12.3(5), 12.6(4), 12.7(2/3), 13.1(3), 13.1.1(4/5), 13.3(2), 13.11.3(3.1/4), 13.12(4.1/2), H.3.2(3), H.7.1(6/5), H.7.1(10/5), J.10(3/2), J.15.1(2/3), J.15.7(2/3), J.15.7(4/3), J.15.14(3/5), J.15.14(5/5), J.15.14(6/5), J.15.15(3/5), J.15.15(4/5), J.15.15(5/5), J.15.15(6/5), L(2.1/5), L(6.1/3), L(10), L(11), L(12.1/5), L(15.1/3), L(16), L(17.1/3), L(26.1/5), L(28.1/5), L(30.1/5), L(31.1/5), L(34.1/5), P.1
in Ada.Direct_IO A.8.4(9), A.8.4(18.12/5), A.8.4(18.5/5)
in Ada.Sequential_IO A.8.1(9), A.8.1(15.12/5), A.8.1(15.5/5)
in Ada.Streams.Stream_IO A.12.1(11), A.12.1(26.12/5), A.12.1(26.5/5)
in Ada.Text_IO A.10.1(12), A.10.1(85.12/5), A.10.1(85.5/5)
in System 13.7(4)
name resolution rules 1.1.2(26/3)
Name_Case_Equivalence
in Ada.Directories A.16(20.2/3)
Name_Case_Kind
in Ada.Directories A.16(20.1/3)
Name_Error
in Ada.Direct_IO A.8.4(18)
in Ada.Directories A.16(43/2)
in Ada.IO_Exceptions A.13(4)
in Ada.Sequential_IO A.8.1(15)
in Ada.Streams.Stream_IO A.12.1(26)
in Ada.Text_IO A.10.1(85/5)
named
in a use clause 8.4(7.1/2)
in a with_clause 10.1.2(6/2)

named association 6.4(7/5), 6.4.1(2/3), 12.3(6)
named component association 4.3.1(6)
named discriminant association 3.7.1(4)
named entry index 9.5.2(21)
named number 3.3(24)
named parameter association 6.4.1(2/3)
named type 3.2.1(7/2)
named_array_aggregate 4.3.3(4/5)
used 4.3.3(2/5), P.1
named_container_aggregate 4.3.5(16/5)
used 4.3.5(13/5), P.1
Names
child of Ada.Interrupts C.3.2(12/5)
Nanoseconds
in Ada.Real_Time D.8(14/2)
Native_Binary
in Interfaces.COBOL B.4(25)
Natural 3.5.4(12)
Natural *subtype of* Integer
in Standard A.1(13)
NBH
in Ada.Characters.Latin_1 A.3.3(17)
NBSP
in Ada.Characters.Latin_1 A.3.3(21/3)
needed
of a compilation unit by another 10.2(2/5)
remote call interface E.2.3(18)
shared passive library unit E.2.1(11)
needed compilation unit 1.3.3(23/5), 10.2(2/5)
needed component 1.3.1(48/5)
extension_aggregate record_component_association_list 4.3.2(6)
record_aggregate record_component_association_list 4.3.1(9/5)
needs finalization 7.6(9.1/2)
language-defined type A.4.5(72.1/2), A.5.2(15.1/2), A.5.2(27.1/2), A.5.6(23/5), A.5.7(26/5), A.8.1(17/2), A.8.4(20/2), A.10.1(86/2), A.12.1(27.1/2), A.16(102/2), A.18.2(147.3/3), A.18.2(84/2), A.18.3(56/2), A.18.3(86.3/3), A.18.4(4/2), A.18.4(41.3/3), A.18.7(4/2), A.18.7(36.4/3), A.18.7(96.2/3), A.18.10(124/3), A.18.10(73/3), A.18.18(27/3), A.18.18(54/3), D.14.2(13/2), D.15(8/2)
NEL
in Ada.Characters.Latin_1 A.3.3(17)
nested declaration 8.1(13/6)
new
See allocator 4.8(1)
New_Char_Array
in Interfaces.C.Strings B.3.1(9)
New_Line
in Ada.Strings.Text_Buffers A.4.12(12/5)
in Ada.Text_IO A.10.1(28/5)
New_Line_Count
in Ada.Strings.Text_Buffers A.4.12(5/5)
New_Page
in Ada.Text_IO A.10.1(31/5)
New_String
in Interfaces.C.Strings B.3.1(10)
New_Vector
in Ada.Containers.Vectors A.18.2(14.1/5)
newly constructed 4.4(9.7/5)
Next
in Ada.Containers.Doubly_Linked_Lists A.18.3(37.1/5), A.18.3(37/5), A.18.3(39.1/5), A.18.3(39/5)
in Ada.Containers.Hashed_Maps A.18.5(28.1/5), A.18.5(28/5), A.18.5(29.1/5), A.18.5(29/5)
in Ada.Containers.Hashed_Sets A.18.8(41.1/5), A.18.8(41/5), A.18.8(42.1/5), A.18.8(42/5)
in Ada.Containers.Ordered_Maps A.18.6(34.1/5), A.18.6(34/5), A.18.6(35.1/5), A.18.6(35/5)
in Ada.Containers.Ordered_Sets A.18.9(45.1/5), A.18.9(45/5), A.18.9(46.1/5), A.18.9(46/5)
in Ada.Containers.Vectors A.18.2(63.1/5), A.18.2(63/5), A.18.2(64.1/5), A.18.2(64/5)
in Ada.Iterator_Interfaces 5.5.1(3/3), 5.5.1(4.7/5)
Next_Sibling
in Ada.Containers.Multiway_Trees A.18.10(64.1/5), A.18.10(64/5), A.18.10(65.1/5), A.18.10(65/5)
No_Abort_Statements restriction D.7(5/3)
No_Access_Parameter_Allocators restriction H.4(8.3/3)
No_Access_Subprograms restriction H.4(17)
No_Allocators restriction H.4(7)
No_Anonymous_Allocators restriction H.4(8.1/3)
No_Break_Space
in Ada.Characters.Latin_1 A.3.3(21/3)
No_Coextensions restriction H.4(8.2/3)
No_Conflict_Checks conflict check policy 9.10.1(22/5)
No_Controlled_Parts aspect H.4.1(2/5)
No_Delay restriction H.4(21)
No_Dependence restriction 13.12.1(6/2)
No_Dispatch restriction H.4(19)
No_Dynamic_Attachment restriction D.7(10/3)
No_Dynamic_CPU_Assignment restriction D.7(10.1/4)
No_Dynamic_Priorities restriction D.7(9/2)
No_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(9/2), A.18.3(51.5/5)
in Ada.Containers.Hashed_Maps A.18.5(6/2), A.18.5(37.6/5)
in Ada.Containers.Hashed_Sets A.18.8(6/2), A.18.8(59.5/5)
in Ada.Containers.Multiway_Trees A.18.10(11/3), A.18.10(70.5/5)
in Ada.Containers.Ordered_Maps A.18.6(7/2), A.18.6(51.7/5)
in Ada.Containers.Ordered_Sets A.18.9(7/2), A.18.9(74.5/5)
in Ada.Containers.Vectors A.18.2(11/2), A.18.2(79.5/5)
No_Exceptions restriction H.4(12/5)
No_Fixed_Point restriction H.4(15)
No_Floating_Point restriction H.4(14)
No_Hidden_Indirect_Globals restriction H.4(23.2/5)
No_Implementation_Aspect_Specifications restriction 13.12.1(1.1/3)
No_Implementation_Attributes restriction 13.12.1(2/6)
No_Implementation_Extensions profile 13.12.1(12/3)
No_Implementation_Identifiers restriction 13.12.1(2.1/3)
No_Implementation_Pragmas restriction 13.12.1(3/2)
No_Implementation_Units restriction 13.12.1(3.1/3)
No_Implicit_Heap_Allocations restriction D.7(8)
No_Index
in Ada.Containers.Vectors A.18.2(7/2)
No_IO restriction H.4(20/5)
No_Local_Allocators restriction H.4(8/1)
No_Local_Protected_Objects restriction D.7(10.2/3)
No_Local_Timing_Events restriction D.7(10.3/3)
No_Nested_Finalization restriction D.7(4/3)
No_Obsolescent_Features restriction 13.12.1(4/3)
No_Parallel_Conflict_Checks conflict check policy 9.10.1(10/5)
No_Protected_Type_Allocators restriction D.7(10.4/2)
No_Protected_Types restriction H.4(5)
No_Recursion restriction H.4(22)
No_Reentrancy restriction H.4(23)
No_Relative_Delay restriction D.7(10.6/3)

No_Requeue_Statements restriction D.7(10.7/3)
No_Return aspect 6.5.1(3.2/3)
No_Return pragma J.15.2(2/5), L(21.2/3)
No_Select_Statements restriction D.7(10.8/3)
No_Specific_Termination_Handlers restriction D.7(10.9/3)
No_Specification_of_Aspect restriction 13.12.1(6.1/3)
No_Standard_Allocators_After_Elaboration restriction D.7(19.2/3)
No_Tag
in Ada.Tags 3.9(6.1/2)
No_Task_Allocators restriction D.7(7)
No_Task_Hierarchy restriction D.7(3/3)
No_Task_Termination restriction D.7(15.1/2)
No_Tasking_Conflict_Checks conflict check policy 9.10.1(11/5)
No_Tasks_Unassigned_To_CPU restriction D.7(10.10/4)
No_Terminate_Alternatives restriction D.7(6)
No_Unchecked_Access restriction H.4(18)
No_Unrecognized_Aspects restriction 13.12.1(6.4/5)
No_Unrecognized_Pragmas restriction 13.12.1(6.5/5)
No_Unspecified_Globals restriction H.4(23.1/5)
No_Use_Of_Attribute restriction 13.12.1(6.2/3)
No_Use_Of_Pragma restriction 13.12.1(6.3/3)
node
of a list A.18.3(2/2)
of a map A.18.4(5/2)
of a tree A.18.10(2/4)
Node_Count
in Ada.Containers.Multiway_Trees A.18.10(17/5)
nominal subtype 1.3.1(49/5), 3.3(23/5), 3.3.1(8/2)
associated with a dereference 4.1(9/3)
associated with a type_conversion 4.6(27)
associated with an indexed_component 4.1.1(5)
of a component 3.6(20)
of a formal parameter 6.1(23/2)
of a function result 6.1(23/2)
of a generic formal object 12.4(9/2)
of a qualified expression 4.7(3.1/3)
of a record component 3.8(14)
of an attribute reference 4.1.4(9/4)
of an enumeration literal 3.5.1(6/3)
of the result of a function_call 6.4(12/2)
nominal type 3.3(23/5)
Non_Preemptive
child of Ada.Dispatching D.2.4(2.2/5)
Non_Preemptive_FIFO_Within_Priorities task disp. policy D.2.4(2/2)
nonblocking 9.5(21/5)
Nonblocking aspect 9.5(19/5)
nonblocking region 9.5(43/5)
nonconfirming
aspect specification 13.1(18.2/5)
operational aspect 13.1(18.2/5)
operational item 13.1(18.2/5)
representation aspect 13.1(18.2/5)
representation item 13.1(18.2/5)
representation value 13.1(18.2/5)
nondispatching call
on a dispatching operation 3.9.2(1/5)
nonexistent 13.11.2(10/4), 13.11.2(16/3)
nongraphic character 4.10(12/5)
nonlimited interface 3.9.4(5/2)
nonlimited type 1.3.1(50/5), 7.5(7)
becoming nonlimited 7.3.1(5/1), 7.5(16)
nonlimited_with_clause 10.1.2(4.2/2)
used 10.1.2(4/2), P.1
nonnormative
See informative 1.1.2(18)
nonoverridable
aspect 13.1.1(18.3/5)
nonoverridable aspect
Aggregate 4.3.5(12/5)
Constant_Indexing 4.1.6(5.1/4)
Default_Iterator 5.5.1(11.1/4)
Implicit_Dereference 4.1.5(5.1/4)
Integer_Literal 4.2.1(6/5)
Iterator_Element 5.5.1(11.1/4)
Max_Entry_Queue_Length D.4(20/5)
No_Controlled_Parts H.4.1(3/5)
Real_Literal 4.2.1(6/5)
String_Literal 4.2.1(6/5)
Variable_Indexing 4.1.6(5.1/4)
nonreturning 6.5.1(3.2/3)
nonstandard integer type 3.5.4(26/5)
nonstandard mode 1.1.5(11)
nonstandard real type 3.5.6(8/5)
normal completion 7.6.1(2/2)
normal library unit E.2(4/3)
normal state of an object 11.6(6/5), 13.9.1(4)
[*partial*] 9.8(21), A.13(17)
Normalize_Scalars pragma H.1(3), L(22)
normalized exponent A.5.3(14)
normalized number A.5.3(10)
normative 1.1.2(14)
not equal operator 4.5(1), 4.5.2(1)
not in (membership test) 4.5(1), 4.5.2(2/3)
not operator 4.5(1), 4.5.6(3)
Not_A_Specific_CPU
in System.Multiprocessors D.16(4/3)
Not_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
notes 1.1.2(39)
Notwithstanding 3.10.2(21.1/5), 7.6(17.5/3), 10.1.6(6/2), 13.1.1(32/6), 13.14(15.3/5), B.1(22/3), B.1(38/3), C.3.1(19/3), E.2.1(8), E.2.1(11), E.2.3(18), H.6(7/2), J.3(6)
[*partial*] J.15.5(8/3)
NUL
in Ada.Characters.Latin_1 A.3.3(5)
in Interfaces.C B.3(20/1)
null access value 4.2(9/5)
null array 3.6.1(7)
null constraint 3.2(7/2)
null extension 3.9.1(4.1/2)
null pointer
See null access value 4.2(9/5)
null procedure 6.7(3/5)
null range 3.5(4)
null record 3.8(15)
null slice 4.1.2(7)
null string literal 2.6(6)
null value
of an access type 3.10(13/2)
Null_Address
in System 13.7(12)
null_array_aggregate 4.3.3(3.1/5)
used 4.3.3(2/5), P.1
Null_Bounded_String
in Ada.Strings.Bounded A.4.4(7)
null_container_aggregate 4.3.5(14/5)
used 4.3.5(13/5), P.1
null_exclusion 3.10(5.1/2)
used 3.2.2(3/2), 3.7(5/5), 3.10(2/2), 3.10(6/2), 6.1(13/2), 6.1(15/5), 8.5.1(2/5), 12.4(2/3), P.1
Null_Id
in Ada.Exceptions 11.4.1(2/5)
Null_Occurrence
in Ada.Exceptions 11.4.1(3/5)
null_procedure_declaration 6.7(2/3)
used 3.1(3/3), 9.4(8/4), P.1
Null_Ptr
in Interfaces.C.Strings B.3.1(7)
Null_Set
in Ada.Strings.Maps A.4.2(5)
in Ada.Strings.Wide_Maps A.4.7(5)
in Ada.Strings.Wide_Wide_Maps A.4.8(5/2)
null_statement 5.1(6)
used 5.1(4/2), P.1
Null_Task_Id
in Ada.Task_Identification C.7.1(2/5)
Null_Unbounded_String
in Ada.Strings.Unbounded A.4.5(5)
number sign 2.1(15/6)

Number_Base *subtype of* Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.5(3/5)
in Ada.Text_IO A.10.1(6)
number_decimal 2.1(10/2)
used 2.3(3.1/3), P.1
number_declaration 3.3.2(2)
used 3.1(3/3), P.1
number_letter 2.1(10.1/2)
used 2.3(3/2), P.1
Number_Of_CPUs
in System.Multiprocessors D.16(5/3)
Number_Sign
in Ada.Characters.Latin_1 A.3.3(8)
numeral 2.4.1(3)
used 2.4.1(2), 2.4.1(4), 2.4.2(3), P.1
Numerator
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(7/5)
Numeric
in Interfaces.COBOL B.4(20/3)
numeric type 3.5(1)
numeric_literal 2.4(2)
used 4.4(7/5), P.1
numerics G(1)
child of Ada A.5(3/5)
Numerics_Assertion_Check 11.5(23.9/5)

O

O(f(N)) A.18(3/2)
object 1.3.1(51/5), 3.3(2)
[*partial*] 3.2(1)
required 5.2(5/2), 6.4.1(5), 12.4(7), 13.1.1(6/3), 13.11(15/5), 13.11.3(4/3)
object-oriented programming (OOP)
See dispatching operations of tagged types 3.9.2(1/5)
See tagged types and type extensions 3.9(1)
object_declaration 3.3.1(2/3)
used 3.1(3/3), 4.5.9(3/5), P.1
object_renaming_declaration 8.5.1(2/5)
used 4.5.9(3/5), 8.5(2), P.1
Object_Size attribute 13.3(58.2/5)
obsolescent feature J(1/2)
occur immediately within 8.1(13/6)
occurrence
of an interrupt C.3(2)
octal
literal 2.4.2(1)
octal literal 2.4.2(1)
Old attribute 6.1.1(26/5)
one's complement
modular types 3.5.4(27)
one-dimensional array 3.6(12)
only as a completion
entry_body 9.5.2(16)
OOP (object-oriented programming)
See dispatching operations of tagged types 3.9.2(1/5)
See tagged types and type extensions 3.9(1)
opaque type
See private types and private extensions 7.3(1)
Open
in Ada.Direct_IO A.8.4(7), A.8.4(18.11/5), A.8.4(18.4/5)
in Ada.Sequential_IO A.8.1(7), A.8.1(15.11/5), A.8.1(15.4/5)
in Ada.Streams.Stream_IO A.12.1(9), A.12.1(26.11/5), A.12.1(26.4/5)
in Ada.Text_IO A.10.1(10), A.10.1(85.11/5), A.10.1(85.4/5)
open alternative 9.7.1(14)
open entry 9.5.3(5)
of a protected object 9.5.3(7/3)
of a task 9.5.3(6/3)
operand
of a qualified_expression 4.7(3/5)
of a type_conversion 4.6(3)
parenthesized expression 4.4(8.1/6)
operand interval G.2.1(6)
operand type
of a type_conversion 4.6(3)
operates on a type 3.2.3(1/2)
operational aspect 1.3.1(53/5), 13.1(8.1/3)
specifiable attributes 13.3(5/3)
operational item 13.1(1.1/1)
operative constituent 4.4(9.2/5)
operator 6.6(1)
& 4.5(1), 4.5.3(3)
* 4.5(1), 4.5.5(1)
** 4.5(1), 4.5.6(7)
+ 4.5(1), 4.5.3(1), 4.5.4(1)
- 4.5(1), 4.5.3(1), 4.5.4(1)
/ 4.5(1), 4.5.5(1)
/= 4.5(1), 4.5.2(1)
< 4.5(1), 4.5.2(1)
<= 4.5(1), 4.5.2(1)
= 4.5(1), 4.5.2(1)
> 4.5(1), 4.5.2(1)
>= 4.5(1), 4.5.2(1)
abs 4.5(1), 4.5.6(1)
ampersand 4.5(1), 4.5.3(3)
and 4.5(1), 4.5.1(2)
binary 4.5(9)
binary adding 4.5.3(1)
concatenation 4.5(1), 4.5.3(3)
divide 4.5(1), 4.5.5(1)
equal 4.5(1), 4.5.2(1)
equality 4.5.2(1)
exponentiation 4.5(1), 4.5.6(7)
greater than 4.5(1), 4.5.2(1)
greater than or equal 4.5(1), 4.5.2(1)
highest precedence 4.5.6(1)
less than 4.5(1), 4.5.2(1)
less than or equal 4.5(1), 4.5.2(1)
logical 4.5.1(2)
minus 4.5(1), 4.5.3(1), 4.5.4(1)
mod 4.5(1), 4.5.5(1)
multiply 4.5(1), 4.5.5(1)
multiplying 4.5.5(1)
not 4.5(1), 4.5.6(3)
not equal 4.5(1), 4.5.2(1)
or 4.5(1), 4.5.1(2)
ordering 4.5.2(1)
plus 4.5(1), 4.5.3(1), 4.5.4(1)
predefined 4.5(9)
relational 4.5.2(1)
rem 4.5(1), 4.5.5(1)
times 4.5(1), 4.5.5(1)
unary 4.5(9)
unary adding 4.5.4(1)
user-defined 6.6(1)
xor 4.5(1), 4.5.1(2)
operator precedence 4.5(1)
operator_symbol 6.1(9)
used 4.1(3), 4.1.3(3), 6.1(5), 6.1(11), P.1
optimization 11.5(29/5), 11.6(1/3)
Optimize pragma 2.8(23), L(23)
or else (short-circuit control form) 4.5(1), 4.5.1(1)
or operator 4.5(1), 4.5.1(2)
Ordered_FIFO_Queuing queuing policy D.4(7.1/5)
Ordered_Maps
child of Ada.Containers A.18.6(2/5)
Ordered_Sets
child of Ada.Containers A.18.9(2/5)
ordering operator 4.5.2(1)
ordinary file A.16(45/2)
ordinary fixed point type 3.5.9(1), 3.5.9(8/2)
ordinary_fixed_point_definition 3.5.9(3)
used 3.5.9(2), P.1
OSC
in Ada.Characters.Latin_1 A.3.3(19)
other_control 2.1(13.1/2)
other_format 2.1(10.3/2)
other_private_use 2.1(13.2/2)
other_surrogate 2.1(13.3/2)
output A.6(1/5)
Output aspect 13.13.2(38/5)
Output attribute 13.13.2(19), 13.13.2(29)
Output clause 13.3(7/2), 13.13.2(38/5)
Output'Class aspect 13.13.2(38/5)
outside
a subtype 3.2(8/5)
overall interpretation
of a complete context 8.6(10)
Overflow_Check 11.5(16)
[*partial*] 3.5.4(20), 4.4(11), G.2.1(11), G.2.2(7), G.2.3(25), G.2.4(2), G.2.6(3/5)

Overlap
in Ada.Containers.Hashed_Sets A.18.8(38/2)
in Ada.Containers.Ordered_Sets A.18.9(39/2)
Overlaps_Storage attribute 13.3(73.6/3)
overload resolution 8.6(1/3)
overloadable 8.3(7)
overloaded 8.3(6)
enumeration literal 3.5.1(9)
overloading rules 1.1.2(26/3), 8.6(2)
overridable 8.3(9/1)
override 8.3(9/1), 12.3(17)
a primitive subprogram 3.2.3(7/2)
when implemented by 9.1(9.2/3), 9.4(11.1/3)
overriding operation 1.3.2(2/5)
overriding_indicator 8.3.1(2/2)
used 3.9.3(1.1/3), 6.1(2/3), 6.3(2/3), 6.7(2/3), 6.8(2/4), 8.5.4(2/3), 9.5.2(2/3), 10.1.3(3/3), 12.3(2/3), P.1
Overwrite
in Ada.Strings.Bounded A.4.4(62), A.4.4(63)
in Ada.Strings.Fixed A.4.3(27), A.4.3(28)
in Ada.Strings.Unbounded A.4.5(57), A.4.5(58)

P

Pack aspect 13.2(5.1/3)
Pack pragma J.15.3(2/3), L(24.1/3)
package 1.3.3(24/5), 7(1)
package instance 12.3(13)
package_body 7.2(2/3)
used 3.11(6), 10.1.1(7), P.1
package_body_stub 10.1.3(4/3)
used 10.1.3(2), P.1
package_declaration 7.1(2)
used 3.1(3/3), 10.1.1(5), P.1
package_renaming_declaration 8.5.3(2/3)
used 8.5(2), 10.1.1(6), P.1
package_specification 7.1(3/3)
used 7.1(2), 12.1(4), P.1
packed 13.2(5.1/3)
Packed_Decimal
in Interfaces.COBOL B.4(12/3)
Packed_Format
in Interfaces.COBOL B.4(26)
Packed_Signed
in Interfaces.COBOL B.4(27)
Packed_Unsigned
in Interfaces.COBOL B.4(27)
padding bits 13.1(7/5)
Page
in Ada.Text_IO A.10.1(39/5)
Page pragma 2.8(22), L(25)
page terminator A.10(7)
Page_Length
in Ada.Text_IO A.10.1(26/5)
Paragraph_Sign
in Ada.Characters.Latin_1 A.3.3(22)
parallel construct 1.3.3(26/5), 5.1(1/5)
parallel iterable container object 5.5.1(11/5)
parallel iterable container type 5.5.1(11/5)
parallel iterator 5.5.2(4/5)
parallel iterator object 5.5.1(6/5)
parallel iterator type 5.5.1(6/5)
parallel loop 5.5(8.1/5)
parallel processing
See parallel construct 5.1(1/5)
See task 9(1/5)
parallel_block_statement 5.6.1(2/5)
used 5.1(5/5), P.1
Parallel_Calls aspect 9.10.1(24/5)
Parallel_Iterator
in Ada.Iterator_Interfaces 5.5.1(4.1/5)
Parallel_Reduce(Reducer, Initial_Value) attribute 4.5.10(33/5)
Parallel_Reversible_Iterator
in Ada.Iterator_Interfaces 5.5.1(4.8/5)
parameter
explicitly aliased 6.1(23.1/3)
See formal parameter 6.1(17)
See generic formal parameter 12(1)
See also discriminant 3.7(1/2)
See also loop parameter 5.5(6/5)
parameter assigning back 6.4.1(17)
parameter copy back 6.4.1(17)
parameter mode 6.1(18/3)
parameter passing 6.4.1(1)
parameter_and_result_profile 6.1(13/2)
used 3.10(5), 3.10(6/2), 6.1(4.2/2), P.1
parameter_association 6.4(5)
used 5.5.3(6/5), 6.4(4), P.1
parameter_association_with_box 5.5.3(7/5)
used 5.5.3(6/5), P.1
parameter_profile 6.1(12)
used 3.10(5), 3.10(6/2), 6.1(4.1/2), 9.5.2(2/3), 9.5.2(3), 9.5.2(6), P.1
parameter_specification 6.1(15/5)
used 6.1(14), P.1
Parameterless_Handler
in Ada.Interrupts C.3.2(2/5)
Params_Stream_Type
in System.RPC E.5(6)
Parent
in Ada.Containers.Multiway_Trees A.18.10(59.1/5), A.18.10(59/5)
parent body
of a subunit 10.1.3(8/2)
parent declaration
of a library unit 10.1.1(10)
of a library_item 10.1.1(10)
parent of a derived type 1.3.1(54/5)
parent subtype 3.4(3/2)
parent type 3.4(3/2)
parent unit
of a library unit 10.1.1(10)
Parent_Tag
in Ada.Tags 3.9(7.2/2)
parent_unit_name 10.1.1(8)
used 6.1(5), 6.1(7), 7.1(3/3), 7.2(2/3), 10.1.3(7), P.1
parenthesized expression 4.4(8.1/6), 4.4(9.1/6)
part
of a type 3.2(6/2)
of an object or value 3.2(6/2)
of the nominal subtype 3.9.1(4.2/5)
partial view 1.3.1(56/5)
of a type 7.3(4)
partition 1.3.3(27/5), 10.2(2/5)
partition building 10.2(2/5)
partition communication subsystem (PCS) E.5(1/2)
Partition_Check
[*partial*] E.4(19)
Partition_Elaboration_Policy pragma H.6(3/2), L(25.1/2)
Partition_Id
in System.RPC E.5(4)
Partition_Id attribute E.1(9)
pass by copy 6.2(2)
pass by reference 6.2(2)
passive partition E.1(2)
Pattern_Error
in Ada.Strings A.4.1(5)
PCS (partition communication subsystem) E.5(1/2)
Peak_Use
in Ada.Containers.Bounded_Priority_Queues A.18.31(7/5)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(6/5)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(7/5)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(7/5)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(6/5)
pending interrupt occurrence C.3(2)
per-object constraint 3.8(18/6)
per-object expression 3.8(18/6)
percent sign 2.1(15/6)
Percent_Sign
in Ada.Characters.Latin_1 A.3.3(8)
perfect result set G.2.3(5)
perform indefinite insertion A.18(6/5)

periodic task
  example 9.6(39)
  See delay_until_statement 9.6(39)
Pi
  in Ada.Numerics A.5(3/5)
Pic_String
  in Ada.Text_IO.Editing F.3.3(7)
Picture
  in Ada.Text_IO.Editing F.3.3(4)
picture String
  for edited output F.3.1(1/3)
Picture_Error
  in Ada.Text_IO.Editing F.3.3(9)
Pilcrow_Sign
  in Ada.Characters.Latin_1 A.3.3(22)
plain_char
  in Interfaces.C B.3(11)
plane
  character 2.1(1/6)
PLD
  in Ada.Characters.Latin_1 A.3.3(17)
PLU
  in Ada.Characters.Latin_1 A.3.3(17)
plus operator 4.5(1), 4.5.3(1), 4.5.4(1)
plus sign 2.1(15/6)
Plus_Minus_Sign
  in Ada.Characters.Latin_1 A.3.3(22)
Plus_Sign
  in Ada.Characters.Latin_1 A.3.3(8)
PM
  in Ada.Characters.Latin_1 A.3.3(19)
point 2.1(15/6)
Pointer
  in Interfaces.C.Pointers B.3.2(5)
  See access value 3.10(1)
  See type System.Address 13.7(34/2)
pointer type
  See access type 3.10(1)
Pointer_Error
  in Interfaces.C.Pointers B.3.2(8)
Pointers
  child of Interfaces.C B.3.2(4/5)
polymorphism 3.9(1), 3.9.2(1/5)
pool
  default 13.11.3(4.2/4)
  subpool 13.11.4(18/3)
pool element 3.10(7/1), 13.11(11)
pool type 13.11(11)
pool-specific access type 3.10(7/1), 3.10(8)
Pool_of_Subpool
  in System.Storage_Pools.Subpools 13.11.4(9/3)
Pos attribute 3.5.5(2)
position 13.5.1(4)
  used 13.5.1(3), P.1
Position attribute 13.5.2(2/2)
position number 3.5(1)
  of an enumeration value 3.5.1(7)
  of an integer value 3.5.4(15)
positional association 6.4(7/5), 6.4.1(2/3), 12.3(6)
positional component association 4.3.1(6)
positional discriminant association 3.7.1(4)
positional parameter association 6.4.1(2/3)
positional_array_aggregate 4.3.3(3/5)
  used 4.3.3(2/5), P.1
positional_container_aggregate 4.3.5(15/5)
  used 4.3.5(13/5), P.1
Positive 3.5.4(12)
Positive *subtype of* Integer
  in Standard A.1(13)
Positive_Count *subtype of* Count
  in Ada.Direct_IO A.8.4(4)
  in Ada.Streams.Stream_IO A.12.1(7)
  in Ada.Text_IO A.10.1(5)
possible interpretation 8.6(14)
  for direct_names 8.3(24)
  for selector_names 8.3(24)
Post aspect 6.1.1(4/5)
Post'Class aspect 6.1.1(5/5)
post-compilation error 1.1.2(29)
post-compilation rules 1.1.2(29)
postcondition 1.3.2(4/5)
postcondition check 6.1.1(35/3)
postcondition expression
  class-wide 6.1.1(5/5)
  explicit 7.3.4(23/5)
  specific 6.1.1(4/5)
potentially blocking operation 9.5(34/5)
  delay_statement 9.6(34), D.9(5)
  remote subprogram call E.4(17/5)
potentially conflict
  actions 9.10(17/5)
potentially signal 9.10(10.1/5)
potentially static expression 6.8(3.2/5)
potentially use-visible 8.4(8/3)
  [*partial*] 12.6(9.2/6)
Pound_Sign
  in Ada.Characters.Latin_1 A.3.3(21/3)
pragma 1.3.3(29/5), 2.8(1), 2.8(2), L(1)
pragma argument 2.8(9)
pragma name 2.8(9)
pragma, aspect-related
  [*partial*] 2.8(7.1/6)
  Attach_Handler J.15.7(3/3)
  CPU J.15.9(1/3)
  Dispatching_Domain J.15.10(1/3)
  Interrupt_Handler J.15.7(1/3)
  Interrupt_Priority J.15.11(3/3)
  Preelaborable_Initialization J.15.14(7/5)
  Priority J.15.11(1/3)
  Relative_Deadline J.15.12(1/3)
  Storage_Size J.15.4(1/3)
  See also pragma, representation 13.1(4/6)
pragma, categorization
  Pure J.15.15(9/5)
  Remote_Call_Interface J.15.15(9/5)
  Remote_Types J.15.15(9/5)
  Shared_Passive J.15.15(9/5)
pragma, configuration 10.1.5(8)
  [*partial*] 2.8(6/6)
  Assertion_Policy 11.4.2(7/3)
  Detect_Blocking H.5(4/2)
  Discard_Names C.5(4)
  Locking_Policy D.3(5)
  Normalize_Scalars H.1(4)
  Partition_Elaboration_Policy H.6(5/2)
  Priority_Specific_Dispatching D.2.2(4/2)
  Profile 13.12(14/3)
  Queuing_Policy D.4(5)
  Restrictions 13.12(8/3)
  Reviewable H.3.1(4)
  Suppress 11.5(5/2)
  Task_Dispatching_Policy D.2.2(4/2)
  Unsuppress 11.5(5/2)
  usable as 2.8(6/6)
pragma, context
  [*partial*] 2.8(7/6)
  Elaborate 10.2.1(23/6)
  Elaborate_All 10.2.1(23/6)
pragma, control
  [*partial*] 2.8(7.3/6)
pragma, executable
  [*partial*] 2.8(7.2/6)
  Assert 11.4.2(4/6)
  Inspection_Point H.3.2(4/6)
pragma, identifier specific to 2.8(10/3)
pragma, interfacing
  Convention J.15.5(1/3)
  Export J.15.5(1/3)
  Import J.15.5(1/3)
pragma, library unit J.15(7/5)
  All_Calls_Remote J.15.15(7/5)
  Elaborate_Body J.15.14(7/5)
  Preelaborate J.15.14(7/5)
  Pure J.15.14(7/5)
  Remote_Call_Interface J.15.15(7/5)
  Remote_Types J.15.15(7/5)
  Shared_Passive J.15.15(7/5)
pragma, program unit J.15(2/5)
  Inline J.15.1(1/3)
  library unit pragmas J.15(7/5)
pragma, representation 13.1(1/1)
  Asynchronous J.15.13(1/3)
  Atomic J.15.8(9/3)
  Atomic_Components J.15.8(9/3)
  Convention J.15.5(1/3)
  Discard_Names C.5(6)
  Export J.15.5(1/3)
  Import J.15.5(1/3)
  Independent J.15.8(9/3)

Independent_Components J.15.8(9/3)
No_Return J.15.2(1/3)
Pack J.15.3(1/3)
Unchecked_Union J.15.6(1/3)
Volatile J.15.8(9/3)
Volatile_Components J.15.8(9/3)
pragma_argument_association 2.8(3/3)
used 2.8(2), 13.12(11/3), L(27.3/3), P.1
pragmas
Admission_Policy D.4.1(3/5), L(1.1/5)
All_Calls_Remote J.15.15(6/5), L(2.1/5)
Assert 11.4.2(3/2), L(2.2/2)
Assertion_Policy 11.4.2(6.1/3), 11.4.2(6/2), L(2.3/2), L(2.4/3)
Asynchronous J.15.13(2/3), L(3.1/3)
Atomic J.15.8(2/3), L(4.1/3)
Atomic_Components J.15.8(5/3), L(5.1/3)
Attach_Handler J.15.7(4/3), L(6.1/3)
Conflict_Check_Policy 9.10.1(3/5), L(6.2/5)
Convention J.15.5(4/3), L(8.1/3)
CPU J.15.9(2/3), L(8.2/3)
Default_Storage_Pool 13.11.3(3/3), L(8.3/3)
Detect_Blocking H.5(3/2), L(8.4/2)
Discard_Names C.5(3), L(9)
Dispatching_Domain J.15.10(2/3), L(9.1/3)
Elaborate 10.2.1(20), L(10)
Elaborate_All 10.2.1(21), L(11)
Elaborate_Body J.15.14(6/5), L(12.1/5)
Export J.15.5(3/3), L(13.1/3)
Generate_Deadlines D.2.6(9.4/5), L(13.2/5)
Import J.15.5(2/3), L(14.1/3)
Independent J.15.8(4/3), L(14.2/3)
Independent_Components J.15.8(7/3), L(14.3/3)
Inline J.15.1(2/3), L(15.1/3)
Inspection_Point H.3.2(3), L(16)
Interrupt_Handler J.15.7(2/3), L(17.1/3)
Interrupt_Priority J.15.11(4/3), L(18.1/3)
Linker_Options B.1(8), L(19)
List 2.8(21), L(20)
Locking_Policy D.3(3), L(21)
No_Return J.15.2(2/5), L(21.2/3)
Normalize_Scalars H.1(3), L(22)
Optimize 2.8(23), L(23)
Pack J.15.3(2/3), L(24.1/3)
Page 2.8(22), L(25)
Partition_Elaboration_Policy H.6(3/2), L(25.1/2)
Preelaborable_Initialization J.15.14(4/5), L(25.3/5)
Preelaborate J.15.14(3/5), L(26.1/5)
Priority J.15.11(2/3), L(27.1/3)
Priority_Specific_Dispatching D.2.2(2.2/2), L(27.2/2)
Profile 13.12(11/3), L(27.3/3)
Pure J.15.14(5/5), L(28.1/5)
Queuing_Policy D.4(3), L(29)
Relative_Deadline J.15.12(2/3), L(29.2/3)
Remote_Call_Interface J.15.15(5/5), L(30.1/5)
Remote_Types J.15.15(4/5), L(31.1/5)
Restrictions 13.12(3), L(32)
Reviewable H.3.1(3), L(33)
Shared_Passive J.15.15(3/5), L(34.1/5)
Storage_Size J.15.4(2/3), L(35.1/3)
Suppress 11.5(4/2), J.10(3/2), L(36)
Task_Dispatching_Policy D.2.2(2), L(37)
Unchecked_Union J.15.6(2/3), L(37.2/3)
Unsuppress 11.5(4.1/2), L(37.3/2)
Volatile J.15.8(3/3), L(38.1/3)
Volatile_Components J.15.8(6/3), L(39.1/3)
Pre aspect 6.1.1(2/5)
Pre'Class aspect 6.1.1(3/5)
precedence of operators 4.5(1)
precondition 1.3.2(5/5)
precondition check
class-wide 6.1.1(33/3)
specific 6.1.1(32/3)
precondition expression
class-wide 6.1.1(3/5)
specific 6.1.1(2/5)
Pred attribute 3.5(25)
predecessor element
of an ordered set A.18.9(81/3)
predecessor node
of an ordered map A.18.6(58/3)
predefined environment A(1)
predefined exception 11.1(4)
predefined library unit
See language-defined library units
predefined operation
of a type 3.2.3(1/2)
predefined operations
of a discrete type 3.5.5(10)
of a fixed point type 3.5.10(17)
of a floating point type 3.5.8(3)
of a record type 3.8(24)
of an access type 3.10.2(34/2)
of an array type 3.6.2(15)
predefined operator 4.5(9)
[*partial*] 3.2.1(9/5)
predefined type 3.2.1(10)
See language-defined types
predicate 4.5.8(3/3)
used 4.5.8(1/3), P.1
predicate aspect 3.2.4(1/5)
predicate check
allocator 3.2.4(31/5)
enabled 3.2.4(7/3)
in out parameters 3.2.4(31/5)
object_declaration 3.2.4(31/5)
subtype conversion 4.6(51/5)
predicate specification 3.2.4(1/5)
predicate-static 3.2.4(15/3)
Predicate_Failure aspect 3.2.4(14.2/4)
predicates satisfied 3.2.4(29.2/4)
predicates satisfied required
membership 4.5.2(29/4)
Valid attribute 13.9.2(3/4), K.2(263/4)
preelaborable
of an elaborable construct 10.2.1(5)
preelaborable initialization 10.2.1(11.1/5)
Preelaborable_Initialization attribute 10.2.1(11.10/6)
Preelaborable_Initialization pragma J.15.14(4/5), L(25.3/5)
Preelaborate aspect 10.2.1(11/5)
Preelaborate pragma J.15.14(3/5), L(26.1/5)
preelaborated 10.2.1(11/5)
[*partial*] 10.2.1(11/5), E.2.1(9)
implementation requirements C.4(2)
preempt
a running task D.2.3(9/2)
preference
for root numeric operators and ranges 8.6(29)
for universal access equality operators 8.6(29.1/3)
preference control
See requeue 9.5.4(1)
prefix 4.1(4)
of a prefixed view 4.1.3(9.2/6)
used 4.1.1(2), 4.1.2(2), 4.1.3(2), 4.1.4(2/5), 4.1.4(4), 4.1.6(10/3), 4.5.10(2/5), 5.5.3(4/5), 6.4(2), 6.4(3), P.1
prefixed view 4.1.3(9.2/6)
prefixed view profile 6.3.1(24.1/2)
Prepend
in Ada.Containers.Doubly_Linked_-Lists A.18.3(22/5)
in Ada.Containers.Vectors A.18.2(45/5)
Prepend_Child
in Ada.Containers.Multiway_Trees A.18.10(51/5)
Prepend_Vector
in Ada.Containers.Vectors A.18.2(44/5)
prescribed result
for the evaluation of a complex arithmetic operation G.1.1(42)
for the evaluation of a complex elementary function G.1.2(35)

for the evaluation of an elementary function A.5.1(37)
Previous
in Ada.Containers.Doubly_Linked_-Lists A.18.3(38.1/5), A.18.3(38/5), A.18.3(40.1/5), A.18.3(40/5)
in Ada.Containers.Ordered_Maps A.18.6(36.1/5), A.18.6(36/5), A.18.6(37.1/5), A.18.6(37/5)
in Ada.Containers.Ordered_Sets A.18.9(47.1/5), A.18.9(47/5), A.18.9(48.1/5), A.18.9(48/5)
in Ada.Containers.Vectors A.18.2(65.1/5), A.18.2(65/5), A.18.2(66.1/5), A.18.2(66/5)
in Ada.Iterator_Interfaces 5.5.1(4/3)
Previous_Sibling
in Ada.Containers.Multiway_Trees A.18.10(66.1/5), A.18.10(66/5), A.18.10(67.1/5), A.18.10(67/5)
primary 4.4(7/5)
used 4.4(6), P.1
primitive
by-reference 13.1(10/5)
primitive function A.5.3(17)
primitive operation
[*partial*] 3.2(1)
primitive operations
of a type 3.2.3(1/2)
primitive operations of a type 1.3.1(58/5)
primitive operator
of a type 3.2.3(8/6)
primitive subprograms
of a type 3.2.3(2)
priority D.1(15/5)
of a protected object D.3(6/2)
Priority aspect D.1(6.2/3)
Priority attribute D.5.2(3/2)
priority inheritance D.1(15/5)
priority inversion D.2.3(11/2)
priority of an entry call D.4(9)
Priority pragma J.15.11(2/3), L(27.1/3)
Priority *subtype of* Any_Priority
in System 13.7(16)
Priority_Queuing queuing policy D.4(8)
Priority_Specific_Dispatching pragma D.2.2(2.2/2), L(27.2/2)
private declaration of a library unit 10.1.1(12)
private descendant
of a library unit 10.1.1(12)
private extension 1.3.1(60/5), 3.2(4.1/2), 3.9(2.1/2), 3.9.1(1/2)
[*partial*] 7.3(14), 12.5.1(5/3)
private library unit 10.1.1(12)
private operations 7.3.1(1)
private part 8.2(5)
of a package 7.1(6/2)
of a protected unit 9.4(11/2)
of a task unit 9.1(9)
private type 1.3.1(61/5), 3.2(4.1/2)
[*partial*] 7.3(14)
private types and private extensions 7.3(1)
private_extension_declaration 7.3(3/3)
used 3.2.1(2), P.1
private_type_declaration 7.3(2/3)
used 3.2.1(2), P.1
procedural_iterator 5.5.3(2/5)
used 5.5(3/5), P.1
procedure 1.3.2(6/5), 6(1)
null 6.7(3/5)
procedure instance 12.3(13)
procedure_call_statement 6.4(2)
used 5.1(4/2), 9.7.2(3.1/2), P.1
procedure_or_entry_call 9.7.2(3.1/2)
used 9.7.2(3/2), 9.7.4(4/2), P.1
procedure_specification 6.1(4.1/2)
used 6.1(4/2), 6.7(2/3), P.1
processing node E(2)
profile 6.1(22)
associated with a dereference 4.1(10)
fully conformant 6.3.1(18/3)
Jorvik D.13(6.1/5)
mode conformant 6.3.1(16/3)
No_Implementation_Extensions 13.12.1(12/3)
Ravenscar D.13(5/3)
subtype conformant 6.3.1(17/3)
type conformant 6.3.1(15/2)
Profile pragma 13.12(11/3), L(27.3/3)
profile resolution rule
name with a given expected profile 8.6(26)
progenitor 1.3.1(63/5)
progenitor subtype 3.9.4(9/2)
progenitor type 3.9.4(9/2)
program 1.3.3(31/5), 10.2(1)
program execution 10.2(1)
program library
See library 10(2)
See library 10.1.4(9/5)
program unit 1.3.3(32/5), 10.1(1)
program unit pragma J.15(2/5)
Inline J.15.1(1/3)
library unit pragmas J.15(7/5)
Program_Error
raised by a nonreturning procedure 6.5.1(9/2)
raised by Address of an intrinsic subprogram 13.3(11.1/3)
raised by barrier failure 9.5.3(7/3)
raised by calling an abstract equality 4.5.2(24.1/5)
raised by closed alternatives 9.7.1(21)
raised by detection of a bounded error 1.1.5(8/5), 4.2.1(17/5), 4.5.10(34/6), 4.8(11.1/2), 5.1(18/5), 5.5.3(28/5), 5.5.3(29/5), 6.2(12/3), 8.5.4(8.1/1), 9.4(20.1/2), 9.5.1(17/5), 9.5.1(18/5), 9.8(20/5), 10.2(26), 11.4.2(23.2/6), 13.9.1(9), 13.11.2(13), 13.11.2(15.2/4), A.18.2(238/3), A.18.2(239.1/3), A.18.2(239.2/3), A.18.2(239/2), A.18.2(247/2), A.18.3(152.1/3), A.18.3(152.2/3), A.18.3(152/2), A.18.4(75.7/3), A.18.4(75.8/3), A.18.7(96.19/3), A.18.7(96.20/3), A.18.10(220/3), A.18.10(221/3), A.18.18(68/3), A.18.18(69/3), A.18.19(10/3), A.18.20(14/3), A.18.21(15/3), A.18.22(12/3), A.18.23(15/3), A.18.24(12/3), A.18.25(14/3), A.18.32(11/5), C.7.1(17/3), D.3(13.7/5), D.10(10.2/5), D.10.1(15/3)
raised by deteection of a bounded error H.5(5/5)
raised by failed finalization 7.6.1(15), 7.6.1(16/5), 7.6.1(17), 7.6.1(17.2/1), 7.6.1(18/2)
raised by failure of runtime check 3.2.4(29.1/4), 3.10.2(29), 3.11(14), 4.6(57/5), 4.8(10.1/6), 4.8(10.2/2), 4.8(10.3/2), 4.8(10.4/3), 5.5(8.1/5), 5.6.1(4/5), 6.5(8/5), 6.5(21/3), 9.5.4(7.2/5), 11.1(4), 11.5(19/5), 12.5.1(23.3/2), 13.3(75.1/6), 13.11.4(27/3), 13.11.4(30/3), A.5.4(13/6), B.3.3(22/2), C.3.1(10/3), C.3.1(11/3), C.3.1(13.1/5), C.3.2(20), D.3(13), D.3(13.5/5), D.4(22/5), D.4(23/5), D.5.2(6/3), D.7(7.1/3), D.7(10.5/3), D.7(19.1/2), E.4(18/1), H.4(23.11/5), J.7.1(7)
raised by finalization of a protected object 9.4(20)
raised for missing return statement 6.4(11/2)
in Standard A.1(46)
Program_Error_Check 11.5(21/5)
[*partial*] 3.2.4(29.1/4), 5.5(8.1/5), 5.6.1(4/5), 12.5.1(23.3/2), 13.3(75.1/6), B.3.3(22/2), H.4(23.11/5)
prohibited
tampering with a holder A.18.18(29/5)
tampering with a list A.18.3(61/5)
tampering with a map A.18.4(7/5)
tampering with a set A.18.7(7/5)
tampering with a tree A.18.10(80/5)
tampering with a vector A.18.2(90/5)
propagate 11.4(1)

an exception occurrence by an execution, to a dynamically enclosing execution 11.4(6)
proper_body 3.11(6)
used 3.11(5), 10.1.3(7), P.1
property
stable 7.3.4(1/5)
property function 7.3.4(2/5)
protected action 9.5.1(4/4)
complete 9.5.1(6)
start 9.5.1(5/4)
protected calling convention 6.3.1(12/4)
protected declaration 9.4(1)
protected entry 9.4(1)
protected function 9.5.1(1)
protected interface 3.9.4(5/2)
protected object 9(3), 9.4(1)
protected operation 9.4(1)
protected procedure 9.5.1(1)
protected subprogram 9.4(1), 9.5.1(1)
protected tagged type 3.9.4(6/2)
protected type 1.3.1(65/5)
protected unit 9.4(1)
protected_body 9.4(7/3)
used 3.11(6), P.1
protected_body_stub 10.1.3(6/3)
used 10.1.3(2), P.1
protected_definition 9.4(4)
used 9.4(2/3), 9.4(3/3), P.1
protected_element_declaration 9.4(6)
used 9.4(4), P.1
protected_operation_declaration 9.4(5/1)
used 9.4(4), 9.4(6), P.1
protected_operation_item 9.4(8/4)
used 9.4(7/3), P.1
protected_type_declaration 9.4(2/3)
used 3.2.1(3/3), P.1
ptrdiff_t
in Interfaces.C B.3(12)
PU1
in Ada.Characters.Latin_1 A.3.3(18)
PU2
in Ada.Characters.Latin_1 A.3.3(18)
public declaration of a library unit 10.1.1(12)
public descendant
of a library unit 10.1.1(12)
public library unit 10.1.1(12)
punctuation_connector 2.1(10.2/2)
used 2.3(3.1/3), P.1
pure 10.2.1(15.1/5)
Pure pragma J.15.14(5/5), L(28.1/5)
pure-barrier-eligible D.7(1.1/5)
Pure_Barriers restriction D.7(10.11/5)
Put
in Ada.Strings.Text_Buffers A.4.12(7/5)
in Ada.Text_IO A.10.1(42/5), A.10.1(48/5), A.10.1(55/5), A.10.1(60/5), A.10.1(66/5), A.10.1(67/5), A.10.1(71/5), A.10.1(72/5), A.10.1(76/5), A.10.1(77/5), A.10.1(82/5), A.10.1(83/5)
in Ada.Text_IO.Bounded_IO A.10.11(4/2), A.10.11(5/2)
in Ada.Text_IO.Complex_IO G.1.3(7), G.1.3(8/5)
in Ada.Text_IO.Editing F.3.3(14/5), F.3.3(15/5), F.3.3(16)
in Ada.Text_IO.Unbounded_IO A.10.12(4/2), A.10.12(5/2)
Put_Image
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(17/5)
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(20/5)
Put_Image aspect 4.10(4/6)
Put_Image attribute 4.10(3/5)
Put_Line
in Ada.Text_IO A.10.1(50/5)
in Ada.Text_IO.Bounded_IO A.10.11(6/2), A.10.11(7/2)
in Ada.Text_IO.Unbounded_IO A.10.12(6/2), A.10.12(7/2)
Put_UTF_8
in Ada.Strings.Text_Buffers A.4.12(10/5)

Q

qualified_expression 4.7(2)
used 4.1(2/5), 4.8(2/3), 13.8(2), P.1
quantified expression 4.5.8(0.1/4)
quantified_expression 4.5.8(1/3)
used 4.4(7/5), P.1
quantifier 4.5.8(2/3)
used 4.5.8(1/3), P.1
Query_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(16.1/5), A.18.3(16/5)
in Ada.Containers.Hashed_Maps A.18.5(16.1/5), A.18.5(16/5)
in Ada.Containers.Hashed_Sets A.18.8(17.1/5), A.18.8(17/5)
in Ada.Containers.Indefinite_Holders A.18.18(14/5)
in Ada.Containers.Multiway_Trees A.18.10(26.1/5), A.18.10(26/5)
in Ada.Containers.Ordered_Maps A.18.6(15.1/5), A.18.6(15/5)
in Ada.Containers.Ordered_Sets A.18.9(16.1/5), A.18.9(16/5)
in Ada.Containers.Vectors A.18.2(31/5), A.18.2(32.1/5), A.18.2(32/5)
Question
in Ada.Characters.Latin_1 A.3.3(10)
Queue
in Ada.Containers.Bounded_Priority_Queues A.18.31(4/6)
in Ada.Containers.Bounded_Synchronized_Queues A.18.29(4/6)
in Ada.Containers.Synchronized_Queue_Interfaces A.18.27(4/3)
in Ada.Containers.Unbounded_Priority_Queues A.18.30(4/3)
in Ada.Containers.Unbounded_Synchronized_Queues A.18.28(4/3)
queueing policy D.4(1/5), D.4(6)
FIFO_Queuing D.4(7/5)
Ordered_FIFO_Queuing D.4(7.1/5)
Priority_Queuing D.4(8)
Queuing_Policy pragma D.4(3), L(29)
Quotation
in Ada.Characters.Latin_1 A.3.3(8)
quotation mark 2.1(15/6)
quoted string
See string_literal 2.6(1)

R

raise
an exception 11(1/3)
an exception 11.3(4/4)
an exception occurrence 11.4(3)
raise an exception 1.3.5(5/5)
Raise_Exception
in Ada.Exceptions 11.4.1(4/3)
raise_expression 11.3(2.1/4)
used 4.4(3/4), P.1
raise_statement 11.3(2/2)
used 5.1(4/2), P.1
Random
in Ada.Numerics.Discrete_Random A.5.2(20/5)
in Ada.Numerics.Float_Random A.5.2(8/5)
random number A.5.2(1)
range 3.5(3), 3.5(4)
of a scalar subtype 3.5(7)
used 3.5(2), 3.6(6), 3.6.1(3), 3.8.1(5/3), 4.4(3.2/4), P.1
Range attribute 3.5(14), 3.6.2(7)
Range(N) attribute 3.6.2(8)
range_attribute_designator 4.1.4(5)

used 4.1.4(4), P.1
range_attribute_reference 4.1.4(4)
used 3.5(3), P.1
Range_Check 11.5(17/5)
[*partial*] 3.2.2(11), 3.5(24), 3.5(27), 3.5(39.12/3), 3.5(39.4/3), 3.5(39.5/3), 3.5(43/5), 3.5(55/5), 3.5.5(7), 3.5.9(19/4), 4.2(11/6), 4.3.3(28), 4.5.1(8), 4.5.6(6), 4.5.6(13), 4.6(28), 4.6(38), 4.6(46), 4.6(51/5), 4.7(4/5), 13.13.2(35/3), A.5.3(26), A.5.3(29), A.5.3(50), A.5.3(53), A.5.3(59), A.5.3(62), K.2(11), K.2(114), K.2(122), K.2(184), K.2(220), K.2(241), K.2(41), K.2(47)
range_constraint 3.5(2)
used 3.2.2(6), 3.5.9(5/4), J.3(2/4), P.1
Ravenscar profile D.13(5/3)
RCI
generic E.2.3(7/5)
library unit E.2.3(7/5)
package E.2.3(7/5)
Re
in Ada.Numerics.Generic_Complex_Arrays G.3.2(7/2), G.3.2(27/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(6)
re-raise statement 11.3(3/4)
read
the value of an object 3.3(14)
in Ada.Direct_IO A.8.4(12/5)
in Ada.Sequential_IO A.8.1(12/5)
in Ada.Storage_IO A.9(6)
in Ada.Streams 13.13.1(5)
in Ada.Streams.Storage.Bounded 13.13.1(28/5)
in Ada.Streams.Storage.Unbounded 13.13.1(20/5)
in Ada.Streams.Stream_IO A.12.1(15/5), A.12.1(16/5)
in System.RPC E.5(7)
Read aspect 13.13.2(38/5)
Read attribute 13.13.2(6), 13.13.2(14)
Read clause 13.3(7/2), 13.13.2(38/5)
Read'Class aspect 13.13.2(38/5)
ready
a task state 9(10/5)
ready queue D.2.1(5/2)
ready task D.2.1(5/2)
Real
in Interfaces.Fortran B.5(6)
real literal 2.4(1)
real literals 3.5.6(4)
real time D.8(18)
real type 1.3.1(66/5), 3.2(3), 3.5.6(1)
real-time systems C(1), D(1/5)
Real_Arrays
child of Ada.Numerics G.3.1(31/2)
Real_Literal aspect 4.2.1(4/5)
Real_Matrix
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
real_range_specification 3.5.7(3)
used 3.5.7(2), 3.5.9(3), 3.5.9(4), P.1
Real_Time
child of Ada D.8(3/5)
real_type_definition 3.5.6(2)
used 3.2.1(4/2), P.1
Real_Vector
in Ada.Numerics.Generic_Real_Arrays G.3.1(4/2)
receiving stub E.4(10)
reclamation of storage 13.11.2(1)
recommended level of support 13.1(20/3)
Address attribute 13.3(15)
Alignment attribute for objects 13.3(33)
Alignment attribute for subtypes 13.3(29)
aspect Pack 13.2(7/3)
bit ordering 13.5.3(7)
Component_Size attribute 13.3(71)
enumeration_representation_clause 13.4(9)
record_representation_clause 13.5.1(17)
required in Systems Programming Annex C.2(2)
Size attribute 13.3(42/5), 13.3(54)
Stream_Size attribute 13.13.2(1.7/2)
unchecked conversion 13.9(16)
with respect to nonstatic expressions 13.1(21/3)
record 3.8(1)
explicitly limited 3.8(13.1/3)
record extension 1.3.1(68/5), 3.4(5/2), 3.9.1(1/2)
Record layout aspect 13.5(1)
record type 1.3.1(69/5), 3.8(1)
record_aggregate 4.3.1(2)
used 4.3(2/5), P.1
record_component_association 4.3.1(4/2)
used 4.3.1(3), P.1
record_component_association_list 4.3.1(3)
used 4.3.1(2), 4.3.2(2), 4.3.4(3/5), P.1
record_definition 3.8(3/5)
used 3.8(2), 3.9.1(2), P.1
record_delta_aggregate 4.3.4(3/5)
used 4.3.4(2/5), P.1
record_extension_part 3.9.1(2)
used 3.4(2/2), P.1
record_representation_clause 13.5.1(2/5)
used 13.1(2/1), P.1
record_type_definition 3.8(2)
used 3.2.1(4/2), P.1
Reduce(Reducer, Initial_Value) attribute 4.5.10(23/5), 4.5.10(32/5)
reducer subprogram 4.5.10(11/5)
reduction expression 1.3.3(34/5), 4.5.10(1/5)
reduction_attribute_designator 4.5.10(4/5)
used 4.5.10(2/5), P.1
reduction_attribute_reference 4.5.10(2/5)
used 4.1.4(2/5), P.1
reduction_specification 4.5.10(5/6)
used 4.5.10(4/5), P.1
reference
dangling 13.11.2(15.1/4)
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.4/5)
in Ada.Containers.Hashed_Maps A.18.5(17.4/5), A.18.5(17.6/5)
in Ada.Containers.Indefinite_Holders A.18.18(19/5)
in Ada.Containers.Multiway_Trees A.18.10(31/5)
in Ada.Containers.Ordered_Maps A.18.6(16.4/5), A.18.6(16.6/5)
in Ada.Containers.Vectors A.18.2(34.4/5), A.18.2(34.6/5)
in Ada.Interrupts C.3.2(10)
in Ada.Task_Attributes C.7.2(5)
reference discriminant 4.1.5(3/3)
reference object 4.1.5(3/3)
reference parameter passing 6.2(2)
reference type 1.3.1(70/5), 4.1.5(3/3)
Reference_Preserving_Key
in Ada.Containers.Hashed_Sets A.18.8(58.2/5), A.18.8(58.4/5)
in Ada.Containers.Ordered_Sets A.18.9(73.2/5), A.18.9(73.4/5)
Reference_Type
in Ada.Containers.Doubly_Linked_Lists A.18.3(17.2/6), A.18.3(51.11/6)
in Ada.Containers.Hashed_Maps A.18.5(17.2/6), A.18.5(37.12/6)
in Ada.Containers.Hashed_Sets A.18.8(58.1/6)
in Ada.Containers.Indefinite_Holders A.18.18(17/6)
in Ada.Containers.Multiway_Trees A.18.10(29/6), A.18.10(70.11/6)
in Ada.Containers.Ordered_Maps A.18.6(16.2/6), A.18.6(51.13/6)
in Ada.Containers.Ordered_Sets A.18.9(73.1/6)
in Ada.Containers.Vectors A.18.2(34.2/6), A.18.2(79.11/6)
references 1.2(1/3)
Registered_Trade_Mark_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Reinitialize
in Ada.Task_Attributes C.7.2(6)
relation 4.4(3/4)
used 4.4(2), P.1
relational operator 4.5.2(1)
relational_operator 4.5(3)

used 4.4(2.2/3), 4.4(3/4), P.1
relative deadline
protected object D.3(13.2/5)
Relative_Deadline aspect D.2.6(9.2/3)
Relative_Deadline attribute D.5.2(3.1/5)
Relative_Deadline pragma J.15.12(2/3), L(29.2/3)
Relative_Deadline *subtype of* Time_Span
in Ada.Dispatching.EDF D.2.6(9/5)
Relative_Name
in
Ada.Directories.Hierarchical_File_Names A.16.1(13/3)
relaxed mode G.2(1)
release
execution resource associated with protected object 9.5.1(6)
rem operator 4.5(1), 4.5.5(1)
Remainder attribute A.5.3(45)
remote access E.1(5)
remote access type E.2.2(9/3)
remote access-to-class-wide type E.2.2(9/3)
remote access-to-subprogram type E.2.2(9/3)
remote call interface E.2(4/3), E.2.3(7/5)
remote procedure call
asynchronous E.4.1(9/3)
remote subprogram E.2.3(7/5)
remote subprogram binding E.4(1/5)
remote subprogram call E.4(1/5)
remote types library unit E.2(4/3), E.2.2(4/5)
Remote_Call_Interface aspect E.2.3(7/5)
Remote_Call_Interface pragma J.15.15(5/5), L(30.1/5)
Remote_Types aspect E.2.2(4/5)
Remote_Types pragma J.15.15(4/5), L(31.1/5)
Remove_Task
in
Ada.Execution_Time.Group_Budgets D.14.2(8/2)
Rename
in Ada.Directories A.16(12/2)
renamed entity 8.5(3)
renamed view 8.5(3)
renaming 1.3.3(35/5)
renaming-as-body 8.5.4(1/3)
renaming-as-declaration 8.5.4(1/3)
renaming_declaration 8.5(2)
used 3.1(3/3), P.1
rendezvous 9.5.2(23.1/5)
repeatedly evaluated
subexpression 6.1.1(22.12/6)
Replace
in Ada.Containers.Hashed_Maps A.18.5(23/5)
in Ada.Containers.Hashed_Sets A.18.8(22/5), A.18.8(53/5)
in Ada.Containers.Ordered_Maps A.18.6(22/5)
in Ada.Containers.Ordered_Sets A.18.9(21/5), A.18.9(66/5)
Replace_Element
in Ada.Containers.Doubly_Linked_Lists A.18.3(15/5)
in Ada.Containers.Hashed_Maps A.18.5(15/5)
in Ada.Containers.Hashed_Sets A.18.8(16/5)
in Ada.Containers.Indefinite_Holders A.18.18(13/5)
in Ada.Containers.Multiway_Trees A.18.10(25/5)
in Ada.Containers.Ordered_Maps A.18.6(14/5)
in Ada.Containers.Ordered_Sets A.18.9(15/5)
in Ada.Containers.Vectors A.18.2(29/5), A.18.2(30/5)
in Ada.Strings.Bounded A.4.4(27)
in Ada.Strings.Unbounded A.4.5(21)
Replace_Slice
in Ada.Containers.Vectors A.18.2(53.3/6), A.18.2(53.4/6)
in Ada.Strings.Bounded A.4.4(58), A.4.4(59)
in Ada.Strings.Fixed A.4.3(23), A.4.3(24)
in Ada.Strings.Unbounded A.4.5(53), A.4.5(54)
Replenish
in
Ada.Execution_Time.Group_Budgets D.14.2(9/2)
Replicate
in Ada.Strings.Bounded A.4.4(78), A.4.4(79), A.4.4(80)
representation
change of 13.6(1/5)
representation aspect 1.3.1(71/5), 13.1(8/5)
coding 13.4(7)
convention, calling convention B.1(1/3)
export B.1(1/3)
external_name B.1(1/3)
import B.1(1/3)
layout 13.5(1)
link_name B.1(1/3)
record layout 13.5(1)
specifiable attributes 13.3(5/3)
storage place 13.5(1)
representation attribute 13.3(1/1)
Bit_Order 13.5.3(3/5)
representation item 13.1(1/1)
representation of an object 13.1(7/5)
representation pragma 13.1(1/1)
Asynchronous J.15.13(1/3)
Atomic J.15.8(9/3)
Atomic_Components J.15.8(9/3)
Convention J.15.5(1/3)
Discard_Names C.5(6)
Export J.15.5(1/3)
Import J.15.5(1/3)
Independent J.15.8(9/3)
Independent_Components J.15.8(9/3)
No_Return J.15.2(1/3)
Pack J.15.3(1/3)
Unchecked_Union J.15.6(1/3)
Volatile J.15.8(9/3)
Volatile_Components J.15.8(9/3)
representation value 13.4(10.4/5), K.2(59.4/5)
representation-oriented attributes
of a fixed point subtype A.5.4(1)
of a floating point subtype A.5.3(1)
representation_clause
See aspect_clause 13.1(4/6)
represented in canonical form A.5.3(10)
requested decimal precision
of a floating point type 3.5.7(4)
requeue 9.5.4(1)
requeue target 9.5.4(3/3)
requeue-with-abort 9.5.4(13)
requeue_statement 9.5.4(2/3)
used 5.1(4/2), P.1
require overriding 3.9.3(6/6)
[*partial*] 6.1.1(16/3)
requires a completion 3.11.1(1/3), 3.11.1(6/3)
declaration for which aspect Elaborate_Body is True 10.2.1(25/6)
declaration of a partial view 7.3(4)
deferred constant declaration 7.4(2/3)
generic_package_declaration 7.1(5/2)
generic_subprogram_declaration 6.1(20/3)
incomplete_type_declaration 3.10.1(3/3)
package_declaration 7.1(5/2)
protected entry_declaration 9.5.2(16)
protected_declaration 9.4(11.2/2)
subprogram_declaration 6.1(20/3)
task_declaration 9.1(9.3/2)
requires late initialization 3.3.1(8.1/5)
Reraise_Occurrence
in Ada.Exceptions 11.4.1(4/3)
Reserve_Capacity
in Ada.Containers.Hashed_Maps A.18.5(9/5)
in Ada.Containers.Hashed_Sets A.18.8(11/5)
in Ada.Containers.Vectors A.18.2(20/5)
reserved interrupt C.3(2)
reserved word 2.9(2/3)
Reserved_128
in Ada.Characters.Latin_1 A.3.3(17)

Reserved_129
  in Ada.Characters.Latin_1 A.3.3(17)
Reserved_132
  in Ada.Characters.Latin_1 A.3.3(17)
Reserved_153
  in Ada.Characters.Latin_1 A.3.3(19)
Reserved_Check
  [*partial*] C.3.1(10/3)
Reset
  in Ada.Direct_IO A.8.4(8)
  in Ada.Numerics.Discrete_Random A.5.2(21/5), A.5.2(24/5)
  in Ada.Numerics.Float_Random A.5.2(9/5), A.5.2(12/5)
  in Ada.Sequential_IO A.8.1(8)
  in Ada.Streams.Stream_IO A.12.1(10)
  in Ada.Text_IO A.10.1(11)
resolution rules 1.1.2(26/3)
resolve
  overload resolution 8.6(14)
restriction 13.12(4/2)
  used 13.12(3), L(32)
restriction_parameter_argument 13.12(4.1/2)
  used 13.12(4/2), P.1
restrictions
  Immediate_Reclamation H.4(10)
  Max_Asynchronous_Select_Nesting D.7(18/1)
  Max_Entry_Queue_Length D.7(19.1/2)
  Max_Image_Length H.4(23.11/5)
  Max_Protected_Entries D.7(14)
  Max_Select_Alternatives D.7(12)
  Max_Storage_At_Blocking D.7(17/1)
  Max_Task_Entries D.7(13)
  Max_Tasks D.7(19/1)
  No_Abort_Statements D.7(5/3)
  No_Access_Parameter_Allocators H.4(8.3/3)
  No_Access_Subprograms H.4(17)
  No_Allocators H.4(7)
  No_Anonymous_Allocators H.4(8.1/3)
  No_Asynchronous_Control J.13(3/2)
  No_Coextensions H.4(8.2/3)
  No_Delay H.4(21)
  No_Dependence 13.12.1(6/2)
  No_Dispatch H.4(19)
  No_Dynamic_Attachment D.7(10/3)
  No_Dynamic_CPU_Assignment D.7(10.1/4)
  No_Dynamic_Priorities D.7(9/2)
  No_Exceptions H.4(12/5)
  No_Fixed_Point H.4(15)
  No_Floating_Point H.4(14)
  No_Hidden_Indirect_Globals H.4(23.2/5)
  No_Implementation_Aspect_Specifications 13.12.1(1.1/3)
  No_Implementation_Attributes 13.12.1(2/6)
  No_Implementation_Identifiers 13.12.1(2.1/3)
  No_Implementation_Pragmas 13.12.1(3/2)
  No_Implementation_Units 13.12.1(3.1/3)
  No_Implicit_Heap_Allocations D.7(8)
  No_IO H.4(20/5)
  No_Local_Allocators H.4(8/1)
  No_Local_Protected_Objects D.7(10.2/3)
  No_Local_Timing_Events D.7(10.3/3)
  No_Nested_Finalization D.7(4/3)
  No_Obsolescent_Features 13.12.1(4/3)
  No_Protected_Type_Allocators D.7(10.4/2)
  No_Protected_Types H.4(5)
  No_Recursion H.4(22)
  No_Reentrancy H.4(23)
  No_Relative_Delay D.7(10.6/3)
  No_Requeue_Statements D.7(10.7/3)
  No_Select_Statements D.7(10.8/3)
  No_Specific_Termination_Handlers D.7(10.9/3)
  No_Specification_of_Aspect 13.12.1(6.1/3)
  No_Standard_Allocators_After_Elaboration D.7(19.2/3)
  No_Task_Allocators D.7(7)
  No_Task_Hierarchy D.7(3/3)
  No_Task_Termination D.7(15.1/2)
  No_Tasks_Unassigned_To_CPU D.7(10.10/4)
  No_Terminate_Alternatives D.7(6)
  No_Unchecked_Access H.4(18)
  No_Unchecked_Conversion J.13(4/2)
  No_Unchecked_Deallocation J.13(5/2)
  No_Unrecognized_Aspects 13.12.1(6.4/5)
  No_Unrecognized_Pragmas 13.12.1(6.5/5)
  No_Unspecified_Globals H.4(23.1/5)
  No_Use_Of_Attribute 13.12.1(6.2/3)
  No_Use_Of_Pragma 13.12.1(6.3/3)
  Pure_Barriers D.7(10.11/5)
  Simple_Barriers D.7(10.12/5)
Restrictions pragma 13.12(3), L(32)
Result attribute 6.1.1(29/5)
result interval
  for a component of the result of evaluating a complex function G.2.6(3/5)
  for the evaluation of a predefined arithmetic operation G.2.1(8)
  for the evaluation of an elementary function G.2.4(2)
result range
  of a Random function A.5.2(40.2/5)
result subtype
  of a function 6.5(3/5)
return expression
  of expression function 6.8(3.1/5)
return object
  extended_return_statement 6.5(5.11/5)
  simple_return_statement 6.5(6/2)
return statement 6.5(1/2)
return_subtype_indication 6.5(2.3/2)
  used 6.5(2.1/5), P.1
returns a readable reference H.4(23.7/5)
returns a writable reference H.4(23.7/5)
reverse iterator 5.5.2(4/5)
Reverse_Elements
  in Ada.Containers.Doubly_Linked_Lists A.18.3(27/5)
  in Ada.Containers.Vectors A.18.2(54/5)
Reverse_Find
  in Ada.Containers.Doubly_Linked_Lists A.18.3(42/5)
  in Ada.Containers.Vectors A.18.2(70/5)
Reverse_Find_Index
  in Ada.Containers.Vectors A.18.2(69/2)
Reverse_Iterate
  in Ada.Containers.Doubly_Linked_Lists A.18.3(46/5)
  in Ada.Containers.Ordered_Maps A.18.6(51/5)
  in Ada.Containers.Ordered_Sets A.18.9(61/5)
  in Ada.Containers.Vectors A.18.2(74/5)
Reverse_Iterate_Children
  in Ada.Containers.Multiway_Trees A.18.10(69.1/5), A.18.10(69/5)
Reverse_Solidus
  in Ada.Characters.Latin_1 A.3.3(12)
reversible iterable container object 5.5.1(11/5)
reversible iterable container type 5.5.1(11/5)
reversible iterator object 5.5.1(6/5)
reversible iterator type 5.5.1(6/5)
Reversible_Iterator
  in Ada.Iterator_Interfaces 5.5.1(4/3)
Reviewable pragma H.3.1(3), L(33)
RI
  in Ada.Characters.Latin_1 A.3.3(17)
right parenthesis 2.1(15/6)
right square bracket 2.1(15/6)
Right_Angle_Quotation
  in Ada.Characters.Latin_1 A.3.3(22)
Right_Curly_Bracket
  in Ada.Characters.Latin_1 A.3.3(14)
Right_Parenthesis
  in Ada.Characters.Latin_1 A.3.3(8)
Right_Square_Bracket
  in Ada.Characters.Latin_1 A.3.3(12)

Ring_Above
in Ada.Characters.Latin_1 A.3.3(22)
root
of a tree A.18.10(2/4)
in Ada.Containers.Multiway_Trees A.18.10(22/5)
root directory A.16(47.1/5)
root library unit 10.1.1(10)
root node
of a tree A.18.10(2/4)
root type
of a class 3.4.1(2/2)
Root_Buffer_Type
in Ada.Strings.Text_Buffers A.4.12(6/5)
root_integer 3.5.4(14)
[*partial*] 3.4.1(8)
root_real 3.5.6(3)
[*partial*] 3.4.1(8)
Root_Storage_Pool
in System.Storage_Pools 13.11(6/5)
Root_Storage_Pool_With_Subpools
in System.Storage_Pools.Subpools 13.11.4(4/5)
Root_Stream_Type
in Ada.Streams 13.13.1(3/2)
Root_Subpool
in System.Storage_Pools.Subpools 13.11.4(5/5)
rooted at a type 3.4.1(2/2)
roots the subtree A.18.10(3/4)
rotate B.2(9)
Rotate_Left B.2(6)
Rotate_Right B.2(6)
Round attribute 3.5.10(12)
Round_Robin
child of Ada.Dispatching D.2.5(4/5)
Round_Robin_Within_Priorities task dispatching policy D.2.5(2/2)
Rounding attribute A.5.3(36), A.5.4(9/6)
RPC
child of System E.5(3/5)
RPC-receiver E.5(21)
RPC_Receiver
in System.RPC E.5(11)
RS
in Ada.Characters.Latin_1 A.3.3(6)
run-time error 1.1.2(30), 1.1.5(6), 11.5(2/3), 11.6(1/3)
run-time polymorphism 3.9.2(1/5)
run-time semantics 1.1.2(30)
run-time type
See tag 3.9(3)
running a program
See program execution 10.2(1)
running task D.2.1(6/2)
runtime check
See language-defined check 11.5(2/3)
runtime name text
entity with C.5(1.1/4)

S

safe range
of a floating point type 3.5.7(9)
of a floating point type 3.5.7(10/5)
Safe_First attribute A.5.3(71), G.2.2(5)
Safe_Last attribute A.5.3(72), G.2.2(6)
safety-critical systems H(1/2)
satisfied
filter 5.5(6.2/5)
satisfies
a discriminant constraint 3.7.1(11)
a range constraint 3.5(4)
an index constraint 3.6.1(7)
for an access value 3.10(15/2)
satisfies the predicates
of a subtype 3.2.4(29.2/4)
Saturday
in Ada.Calendar.Formatting 9.6.1(17/2)
Save
in Ada.Numerics.Discrete_Random A.5.2(24/5)
in Ada.Numerics.Float_Random A.5.2(12/5)
Save_Occurrence
in Ada.Exceptions 11.4.1(6/2)
scalar type 1.3.1(72/5), 3.2(3), 3.5(1)
scalar_constraint 3.2.2(6)
used 3.2.2(5), P.1
scale
of a decimal fixed point subtype 3.5.10(11), K.2(216)
Scale attribute 3.5.10(11)
Scaling attribute A.5.3(27)
SCHAR_MAX
in Interfaces.C B.3(6)
SCHAR_MIN
in Interfaces.C B.3(6)
SCI
in Ada.Characters.Latin_1 A.3.3(19)
scope
informal definition 3.1(8)
of (a view of) an entity 8.2(11)
of a declaration 8.2(10)
of a use_clause 8.4(6)
of a with_clause 10.1.2(5)
of an aspect_specification 8.2(10.1/3)
of an attribute_definition_clause 8.2(10.1/3)
Search_Type
in Ada.Directories A.16(31/2)
Second
in Ada.Calendar.Formatting 9.6.1(26/2)
Second_Duration *subtype of* Day_Duration
in Ada.Calendar.Formatting 9.6.1(20/2)
Second_Number *subtype of* Natural
in Ada.Calendar.Formatting 9.6.1(20/2)
Seconds
in Ada.Calendar 9.6(13)
in Ada.Real_Time D.8(14/2)
Seconds_Count
in Ada.Real_Time D.8(15)
Seconds_Of
in Ada.Calendar.Formatting 9.6.1(23.1/6), 9.6.1(28/2)
Section_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
secure systems H(1/2)
select an entry call
from an entry queue 9.5.3(13), 9.5.3(16)
immediately 9.5.3(8)
select_alternative 9.7.1(4)
used 9.7.1(2), P.1
select_statement 9.7(2)
used 5.1(5/5), P.1
selected_component 4.1.3(2)
used 4.1(2/5), P.1
selection
of an entry caller 9.5.2(24/5)
selective_accept 9.7.1(2)
used 9.7(2), P.1
selector_name 4.1.3(3)
used 3.7.1(3/5), 4.1.3(2), 4.3.1(5/5), 5.5.3(7/5), 6.4(5), 12.3(4), 12.7(3.1/2), P.1
semantic dependence
of one compilation unit upon another 10.1.1(26/2)
semicolon 2.1(15/6)
in Ada.Characters.Latin_1 A.3.3(10)
separate compilation 10.1(1)
Separate_Interrupt_Clocks_Supported
in Ada.Execution_Time D.14(9.2/3)
separator 2.2(3/2)
separator_line 2.1(12/2)
separator_paragraph 2.1(12.1/2)
separator_space 2.1(11/2)
sequence of characters
of a string_literal 2.6(5)
sequence_of_statements 5.1(2/3)
used 5.3(2), 5.4(3), 5.5(2), 5.6.1(2/5), 9.7.1(2), 9.7.1(5), 9.7.1(6), 9.7.2(3/2), 9.7.3(2), 9.7.4(3), 9.7.4(5), 11.2(2), 11.2(3/5), P.1
sequential
actions 9.10(11/5), C.6(17)
sequential access A.8(2)
sequential file A.8(1/2)
sequential iterator 5.5.2(4/5)
sequential loop 5.5(9/5)
Sequential_IO
child of Ada A.8.1(2/5)

service
an entry queue 9.5.3(13)
set
execution timer object D.14.1(12/2)
group budget object D.14.2(15/2)
termination handler C.7.3(9/2)
timing event object D.15(9/2)
in Ada.Containers.Hashed_Sets A.18.8(3/5), A.18.8(59.2/5)
in Ada.Containers.Ordered_Sets A.18.9(4/5), A.18.9(74.2/5)
in Ada.Environment_Variables A.17(6/2)
set container A.18.7(1/2)
Set_Bounded_String
in Ada.Strings.Bounded A.4.4(12.1/2)
Set_Col
in Ada.Text_IO A.10.1(35/5)
Set_CPU
in System.Multiprocessors.Dispatching_Domains D.16.1(12/3)
Set_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Set_Dependents_Fallback_Handler
in Ada.Task_Termination C.7.3(5/2)
Set_Directory
in Ada.Directories A.16(6/2)
Set_Error
in Ada.Text_IO A.10.1(15)
Set_Exit_Status
in Ada.Command_Line A.15(9)
Set_False
in Ada.Synchronous_Task_Control D.10(4/5)
Set_Handler
in Ada.Execution_Time.Group_Budgets D.14.2(10/2)
in Ada.Execution_Time.Timers D.14.1(7/2)
in Ada.Real_Time.Timing_Events D.15(5/2)
Set_Im
in Ada.Numerics.Generic_Complex_Arrays G.3.2(8/2), G.3.2(28/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(7)
Set_Index
in Ada.Direct_IO A.8.4(14/5)
in Ada.Streams.Stream_IO A.12.1(22/5)
Set_Input
in Ada.Text_IO A.10.1(15)
Set_Iterator_Interfaces
in Ada.Containers.Hashed_Sets A.18.8(6.3/3), A.18.8(59.7/5)
in Ada.Containers.Ordered_Sets A.18.9(7.3/3), A.18.9(74.7/5)
Set_Length
in Ada.Containers.Vectors A.18.2(22/5)
Set_Line
in Ada.Text_IO A.10.1(36/5)
Set_Line_Length
in Ada.Text_IO A.10.1(23/5)
Set_Mode
in Ada.Streams.Stream_IO A.12.1(24)
Set_Output
in Ada.Text_IO A.10.1(15)
Set_Page_Length
in Ada.Text_IO A.10.1(24/5)
Set_Pool_of_Subpool
in System.Storage_Pools.Subpools 13.11.4(10/5)
Set_Priority
in Ada.Dynamic_Priorities D.5.1(4)
Set_Quantum
in Ada.Dispatching.Round_Robin D.2.5(4/5)
Set_Re
in Ada.Numerics.Generic_Complex_Arrays G.3.2(8/2), G.3.2(28/2)
in Ada.Numerics.Generic_Complex_Types G.1.1(7)
Set_Relative_Deadline
in Ada.Dispatching.EDF D.2.6(9/5)
Set_Specific_Handler
in Ada.Task_Termination C.7.3(6/2)
Set_True
in Ada.Synchronous_Task_Control D.10(4/5)
Set_Unbounded_String
in Ada.Strings.Unbounded A.4.5(11.1/2)
Set_Value
in Ada.Task_Attributes C.7.2(6)
shared passive library unit E.2(4/3), E.2.1(4/5)
shared variable
protection of 9.10(1/5)
Shared_Passive aspect E.2.1(4/5)
Shared_Passive pragma J.15.15(3/5), L(34.1/5)
shift B.2(9)
Shift_Left B.2(6)
Shift_Right B.2(6)
Shift_Right_Arithmetic B.2(6)
short
in Interfaces.C B.3(7)
short-circuit control form 4.5.1(1)
Short_Float 3.5.7(16/5)
Short_Integer 3.5.4(25)
SI
in Ada.Characters.Latin_1 A.3.3(5)
signal
as defined between actions 9.10(2/5)
See interrupt C.3(1/3)
signal (an exception)
See raise 11(1/3)
signal handling
example 9.7.4(10)
signed integer type 3.5.4(1)
signed_char
in Interfaces.C B.3(8)
Signed_Conversions
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(12/5)
signed_integer_type_definition 3.5.4(3)
used 3.5.4(2), P.1
Signed_Zeros attribute A.5.3(13)
simple entry call 9.5.3(1)
simple name
of a file A.16(47/2)
Simple_Barriers restriction D.7(10.12/5)
simple_expression 4.4(4)
used 3.5(3), 3.5.4(3), 3.5.7(3), 3.5.9(5/4), 4.4(2.2/3), 4.4(3.2/4), 4.4(3/4), 5.5(3.1/5), 11.3(2.1/4), 13.5.1(5), 13.5.1(6), J.3(2/4), P.1
Simple_Name
in Ada.Directories A.16(16/5), A.16(38/2)
in Ada.Directories.Hierarchical_File_Names A.16.1(10/3)
simple_return_statement 6.5(2/2)
used 5.1(4/2), P.1
simple_statement 5.1(4/2)
used 5.1(3), P.1
Sin
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(4)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(5)
single
class expected type 8.6(27/6)
single entry 9.5.2(20)
Single_Precision_Complex_Types
in Interfaces.Fortran B.5(8)
single_protected_declaration 9.4(3/3)
used 3.3.1(2/3), P.1
single_task_declaration 9.1(3/5)
used 3.3.1(2/3), P.1
Sinh
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(6)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(7)
size
of an object 13.1(7/5)
in Ada.Direct_IO A.8.4(15)
in Ada.Directories A.16(26/2), A.16(41/2)
in Ada.Streams.Stream_IO A.12.1(23)
Size (object) aspect 13.3(41)
Size (subtype) aspect 13.3(48)

Size attribute 13.3(40), 13.3(45)
Size clause 13.3(7/2), 13.3(41), 13.3(48)
size_t
in Interfaces.C B.3(13)
Skip_Line
in Ada.Text_IO A.10.1(29/5)
Skip_Page
in Ada.Text_IO A.10.1(32/5)
slice 4.1.2(2)
used 4.1(2/5), P.1
in Ada.Containers.Vectors A.18.2(53.1/6), A.18.2(53.2/6)
in Ada.Strings.Bounded A.4.4(28)
in Ada.Strings.Unbounded A.4.5(22)
small
of a fixed point type 3.5.9(8/2)
Small aspect 3.5.10(2/5)
Small attribute 3.5.10(2/5)
Small clause 3.5.10(2/5), 13.3(7/2)
smallest first order A.18(5/5)
SO
in Ada.Characters.Latin_1 A.3.3(5)
Soft_Hyphen
in Ada.Characters.Latin_1 A.3.3(21/3)
SOH
in Ada.Characters.Latin_1 A.3.3(5)
solidus 2.1(15/6)
in Ada.Characters.Latin_1 A.3.3(8)
Solve
in Ada.Numerics.Generic_Complex_Arrays G.3.2(46/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(24/2)
Sort
in Ada.Containers.Doubly_Linked_Lists A.18.3(49/5)
in Ada.Containers.Vectors A.18.2(77/5)
SOS
in Ada.Characters.Latin_1 A.3.3(19)
SPA
in Ada.Characters.Latin_1 A.3.3(18)
Space
in Ada.Characters.Latin_1 A.3.3(8)
in Ada.Strings A.4.1(4/2)
special file A.16(45/2)
special graphic character
a category of Character A.3.2(32)
Special_Set
in Ada.Strings.Maps.Constants A.4.6(4)
specialized needs annex 1.3.3(36/5)
Specialized Needs Annexes 1.1.2(7)
specifiable
of Address for entries J.7.1(6)
of Address for stand-alone objects and for program units 13.3(12)
of Alignment for first subtypes 13.3(26.4/2)
of Alignment for objects 13.3(25/2)
of Bit_Order for record types and record extensions 13.5.3(4)
of Component_Size for array types 13.3(70)
of External_Tag for a tagged type 13.3(75/3), K.2(65)
of Input for a type 13.13.2(38/5)
of Machine_Radix for decimal first subtypes F.1(1/5)
of Output for a type 13.13.2(38/5)
of Read for a type 13.13.2(38/5)
of Size for first subtypes 13.3(48)
of Size for stand-alone objects 13.3(41)
of Small for fixed point types 3.5.10(2/5)
of Storage_Pool for a nonderived access-to-object type 13.11(15/5)
of Storage_Size for a nonderived access-to-object type 13.11(15/5)
of Storage_Size for a task first subtype J.9(3/3)
of Write for a type 13.13.2(38/5)
specifiable (of an attribute and for an entity) 13.3(5/3)
specific handler C.7.3(9/2)
specific postcondition expression 6.1.1(4/5)
specific precondition expression 6.1.1(2/5)
specific stable property function 7.3.4(5/5)
specific type 3.4.1(3/5)
Specific_Handler
in Ada.Task_Termination C.7.3(6/2)
specified
of an aspect of representation of an entity 13.1(17)
of an operational aspect of an entity 13.1(18.1/5)
specified (not!) 1.1.3(18)
specified as independently addressable C.6(8.1/4)
specified discriminant 3.7(18)
Splice
in Ada.Containers.Doubly_Linked_Lists A.18.3(30/5), A.18.3(31/5), A.18.3(32/2)
Splice_Children
in Ada.Containers.Multiway_Trees A.18.10(57/5), A.18.10(58/5)
Splice_Subtree
in Ada.Containers.Multiway_Trees A.18.10(55/5), A.18.10(56/5)
Split
in Ada.Calendar 9.6(14)
in Ada.Calendar.Formatting 9.6.1(29/2), 9.6.1(32/2), 9.6.1(33/2), 9.6.1(34/2)
in Ada.Execution_Time D.14(8/2)
in Ada.Real_Time D.8(16)
Split_Into_Chunks
in Ada.Iterator_Interfaces 5.5.1(4.4/5)
Sqrt
in Ada.Numerics.Generic_Complex_Elementary_Functions G.1.2(3)
in Ada.Numerics.Generic_Elementary_Functions A.5.1(4)
SS2
in Ada.Characters.Latin_1 A.3.3(17)
SS3
in Ada.Characters.Latin_1 A.3.3(17)
SSA
in Ada.Characters.Latin_1 A.3.3(17)
ST
in Ada.Characters.Latin_1 A.3.3(19)
stabilized view
list A.18.3(151.1/5)
map A.18.4(75.1/5)
set A.18.7(96.13/5)
tree A.18.10(219.1/5)
vector A.18.2(237.1/5)
stable
list A.18.3(151.1/5)
map A.18.4(75.1/5)
set A.18.7(96.13/5)
tree A.18.10(219.1/5)
vector A.18.2(237.1/5)
in Ada.Containers.Doubly_Linked_Lists A.18.3(51.1/5)
in Ada.Containers.Hashed_Maps A.18.5(37.2/5)
in Ada.Containers.Hashed_Sets A.18.8(59.1/5)
in Ada.Containers.Multiway_Trees A.18.10(70.1/5)
in Ada.Containers.Ordered_Maps A.18.6(51.3/5)
in Ada.Containers.Ordered_Sets A.18.9(74.1/5)
in Ada.Containers.Vectors A.18.2(79.1/5)
stable property 1.3.1(73/5), 7.3.4(1/5)
stable property function
class-wide 7.3.4(6/5)
of a type 7.3.4(7/5)
specific 7.3.4(5/5)
Stable_Properties aspect 7.3.4(5/5)
Stable_Properties'Class aspect 7.3.4(6/5)
stand-alone constant 3.3.1(23/5)
corresponding to a formal object of mode in 12.4(10/2)
stand-alone object 3.3.1(1/3)
[*partial*] 12.4(10/2)
stand-alone variable 3.3.1(23/5)
Standard A.1(4/5)
standard error file A.10(6)
standard input file A.10(5)
standard mode 1.1.5(11)
standard output file A.10(5)
standard storage pool 13.11(17)

Standard_Error
  in Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Indent
  in Ada.Strings.Text_Buffers A.4.12(13/5)
Standard_Input
  in Ada.Text_IO A.10.1(16), A.10.1(19)
Standard_Output
  in Ada.Text_IO A.10.1(16), A.10.1(19)
Start_Search
  in Ada.Directories A.16(32/2)
State
  in Ada.Numerics.Discrete_Random A.5.2(23)
  in Ada.Numerics.Float_Random A.5.2(11)
statement 5.1(3)
  used 5.1(2/3), P.1
statement_identifier 5.1(8)
  used 5.1(7), 5.5(2), 5.6(2), P.1
static 4.9(1)
  constant 4.9(24/5)
  constraint 4.9(27)
  delta constraint 4.9(29)
  digits constraint 4.9(29)
  discrete_range 4.9(25)
  discriminant constraint 4.9(31)
  expression 4.9(2)
  function 4.9(18)
  index constraint 4.9(30)
  range 4.9(25)
  range constraint 4.9(29)
  scalar subtype 4.9(26/3)
  string subtype 4.9(26/3)
  subtype 4.9(26/3)
  subtype 12.4(9/2)
Static aspect 6.8(3.6/5)
static expression
  required 3.3.2(4/5), 3.5(56.3/3), 3.5.4(6), 3.5.4(7), 3.5.7(6), 3.5.9(7), 3.6(22.2/3), 3.6.2(1), 3.8.1(8/3), 4.1.4(8), 5.4(5/3), 5.5.3(18/5), 5.5.3(20/5), 6.3.2(5.2/3), 6.5.1(3.3/5), 13.1.1(32/6), 13.2(5.2/3), 13.3(25/2), 13.3(26.4/2), 13.3(41), 13.3(48), 13.3(58.2/5), 13.3(65.4/6), 13.3(70), 13.3(75/3), 13.4(6/2), 13.5.1(10), 13.12(6), 13.13.2(1.5/2), B.1(27/3), B.3.3(3.2/3), C.3.1(6.2/3), C.5(1.3/4), C.6(6.11/5), D.1(8/3), D.2.1(1.8/5), D.2.2(3.2/2), D.2.6(9.6/5), D.4(19/5), D.16(9/3), E.4.1(8.1/3), E.4.1(8.3/3), F.1(1/5), H.4.1(2/5), J.3(4/4), J.15.11(7/3), K.2(65)
static expression function 6.8(3.6/5)
static semantics 1.1.2(28)
Static_Predicate aspect 3.2.4(1/5)
statically compatible
  for a constraint and a scalar subtype 4.9.1(4/3)
  for a constraint and an access or composite subtype 4.9.1(4/3)
  for two subtypes 4.9.1(5/3)
  required 3.7(15), 6.5(5.3/5), 7.4(6/6), 8.5.1(4.10/5), 12.5.1(7/5)
statically constrained 4.9(32)
statically deeper 3.10.2(4), 3.10.2(17)
statically denote 4.9(14)
statically determined tag 3.9.2(1/5)
  [*partial*] 3.9.2(15), 3.9.2(19)
statically match
  for global aspects 4.9.1(1.5/5)
statically matching
  effect on subtype-specific aspects 13.1(14/5)
  for constraints 4.9.1(1/2)
  for ranges 4.9.1(3)
  for subtypes 4.9.1(2/5)
  required 3.9.2(10/2), 3.10.2(27.1/2), 4.6(24.16/6), 4.6(24.5/2), 6.3.1(16.3/3), 6.3.1(17/3), 6.3.1(23), 6.5(5.3/5), 6.5(5.4/5), 7.3(13), 8.5.1(4.2/2), 12.4(8.1/2), 12.5.1(14), 12.5.3(6), 12.5.3(7), 12.5.4(3), 12.7(7)
statically names 4.9(17.1/5)
statically tagged 3.9.2(4/2)
statically unevaluated 4.9(32.1/3)
Status_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Storage
  child of Ada.Streams 13.13.1(12/5)
storage deallocation
  unchecked 13.11.2(1)
storage element 13.3(8)
storage management
  user-defined 13.11(1)
storage node E(2)
storage place
  of a component 13.5(1)
  representation aspect 13.5(1)
storage place attributes
  of a component 13.5.2(1)
storage pool 3.10(7/1)
  default 13.11.3(4.2/4)
storage pool element 13.11(11)
storage pool object 1.3.1(74/5)
storage pool that supports subpools 13.11.4(18/3)
storage pool type 13.11(11)
Storage_Array
  in System.Storage_Elements 13.7.1(5)
Storage_Check 11.5(23)
  [*partial*] 11.1(6/5), 13.3(67), 13.11(17), D.7(17/1), D.7(18/1), D.7(19/1)
Storage_Count *subtype of* Storage_Offset
  in System.Storage_Elements 13.7.1(4)
Storage_Element
  in System.Storage_Elements 13.7.1(5)
Storage_Elements
  child of System 13.7.1(2/5)
Storage_Error
  raised by detection of a bounded error 8.5.4(8.1/1)
  raised by failure of runtime check 11.1(4), 11.1(6/5), 11.5(23), 13.3(67), 13.11(17), 13.11(18/6), D.7(17/1), D.7(18/1), D.7(19.3/3), D.7(19/1)
  in Standard A.1(46)
Storage_IO
  child of Ada A.9(3/5)
Storage_Offset
  in System.Storage_Elements 13.7.1(3)
Storage_Pool aspect 13.11(15/5)
Storage_Pool attribute 13.11(13)
Storage_Pool clause 13.3(7/2), 13.11(15/5)
storage_pool_indicator 13.11.3(3.1/4)
  used 13.11.3(3/3), L(8.3/3)
Storage_Pools
  child of System 13.11(5/5)
Storage_Size
  in System.Storage_Pools 13.11(9)
  in System.Storage_Pools.Subpools 13.11.4(16/3)
Storage_Size (access) aspect 13.11(15/5)
Storage_Size (task) aspect 13.3(65.2/3)
Storage_Size attribute 13.3(60/3), 13.11(14), J.9(2)
Storage_Size clause 13.3(7/2), 13.11(15/5)
Storage_Size pragma J.15.4(2/3), L(35.1/3)
Storage_Stream_Type
  in Ada.Streams.Storage 13.13.1(13/5)
Storage_Unit
  in System 13.7(13)
stream 1.3.1(76/5), 13.13(1)
  in Ada.Streams.Stream_IO A.12.1(13)
  in Ada.Text_IO.Text_Streams A.12.2(4)
  in Ada.Wide_Text_IO.Text_Streams A.12.3(4)
  in Ada.Wide_Wide_Text_IO.Text_-Streams A.12.4(4/2)
stream file A.8(1/2)
stream type 13.13(1)
stream-oriented attributes 13.13.2(1/3)
Stream_Access
  in Ada.Streams.Stream_IO A.12.1(4)
  in Ada.Text_IO.Text_Streams A.12.2(3/5)

in Ada.Wide_Text_IO.Text_Streams A.12.3(3/5)
in Ada.Wide_Wide_Text_IO.Text_-Streams A.12.4(3/5)
Stream_Element
in Ada.Streams 13.13.1(4/1)
Stream_Element_Array
in Ada.Streams 13.13.1(4/1)
Stream_Element_Count *subtype of* Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_Element_Offset
in Ada.Streams 13.13.1(4/1)
Stream_IO
child of Ada.Streams A.12.1(3/5)
Stream_Size aspect 13.13.2(1.5/2)
Stream_Size attribute 13.13.2(1.2/3)
Stream_Size clause 13.3(7/2)
Stream_Type
in Ada.Streams.Storage.Bounded 13.13.1(27/5)
in Ada.Streams.Storage.Unbounded 13.13.1(19/5)
Streams
child of Ada 13.13.1(2/5)
strict mode G.2(1)
strict weak ordering A.18(5/5)
String
in Standard A.1(37/3)
string type 3.6.3(1)
String_Access
in Ada.Strings.Unbounded A.4.5(7)
string_element 2.6(3)
used 2.6(2), P.1
string_literal 2.6(2)
used 4.4(7/5), 6.1(9), P.1
String_Literal aspect 4.2.1(5/5)
Strings
child of Ada A.4.1(3/5)
child of Ada.Strings.UTF_Encoding A.4.11(22/5)
child of Interfaces.C B.3.1(3/5)
Strings_Assertion_Check 11.5(23.10/5)
Strlen
in Interfaces.C.Strings B.3.1(17)
structure
See record type 3.8(1)
STS
in Ada.Characters.Latin_1 A.3.3(18)
STX
in Ada.Characters.Latin_1 A.3.3(5)
SUB
in Ada.Characters.Latin_1 A.3.3(6)
Sub_Second
in Ada.Calendar.Formatting 9.6.1(27/2)
subaggregate
of an array_aggregate 4.3.3(6)
subcomponent 3.2(6/2)
of the nominal subtype 3.9.1(4.2/5)
subpool 13.11.4(18/3)
subpool access type 13.11.4(22/3)
subpool handle 13.11.4(18/3)
Subpool_Handle
in System.Storage_Pools.Subpools 13.11.4(6/3)
subpool_specification 4.8(2.1/3)
used 4.8(2/3), P.1
Subpools
child of System.Storage_Pools 13.11.4(3/5)
subprogram 1.3.2(7/5), 6(1)
abstract 3.9.3(3/2)
allows exit 5.5.3(19/5)
subprogram call 6.4(1)
subprogram instance 12.3(13)
subprogram property aspect definition 7.3.4(3/5)
subprogram_body 6.3(2/3)
used 3.11(6), 9.4(8/4), 10.1.1(7), P.1
subprogram_body_stub 10.1.3(3/3)
used 10.1.3(2), P.1
subprogram_declaration 6.1(2/3)
used 3.1(3/3), 9.4(5/1), 9.4(8/4), 10.1.1(5), P.1
subprogram_default 12.6(3/2)
used 12.6(2.1/3), 12.6(2.2/3), P.1
subprogram_renaming_declaration 8.5.4(2/3)
used 8.5(2), 10.1.1(6), P.1
subprogram_specification 6.1(4/2)
used 3.9.3(1.1/3), 6.1(2/3), 6.3(2/3), 8.5.4(2/3), 10.1.3(3/3), 12.1(3/3), 12.6(2.1/3), 12.6(2.2/3), P.1
subsequence
reduction attribute 4.5.10(21/6)
subsystem 10.1(3)
subtree
node which roots A.18.10(3/4)
of a tree A.18.10(3/4)
Subtree_Node_Count
in Ada.Containers.Multiway_Trees A.18.10(18.1/5), A.18.10(18/5)
subtype 1.3.1(77/5), 3.2(8/5)
constraint of 3.2(8/5)
type of 3.2(8/5)
values belonging to 3.2(8/5)
values outside of 3.2(8/5)
subtype (of an object)
See actual subtype of an object 3.3(23/5)
See actual subtype of an object 3.3.1(9/2)
subtype conformance 6.3.1(17/3)
[*partial*] 3.10.2(34/2), 9.5.4(17/5)
required 3.9.2(10/2), 3.10.2(32/5), 4.5.2(9.8/6), 4.6(24.24/6), 8.5.1(4.3/2), 8.5.4(5/3), 9.1(9.7/2), 9.1(9.8/2), 9.4(11.6/2), 9.4(11.7/2), 9.5.4(5/3), 12.4(8.2/2), 12.5.4(5/3)
subtype conversion
See type conversion 4.6(1/3)
See also implicit subtype conversion 4.6(1/3)
subtype-specific
of a representation item 13.1(8/5)
of an aspect 13.1(8/5)
subtype_declaration 3.2.2(2/3)
used 3.1(3/3), P.1
subtype_indication 3.2.2(3/2)
used 3.2.2(2/3), 3.3.1(2/3), 3.4(2/2), 3.6(6), 3.6(7/2), 3.6.1(3), 3.8.1(5/3), 3.10(3), 4.8(2/3), 5.5.2(2.1/5), 6.5(2.3/2), 7.3(3/3), P.1
subtype_mark 3.2.2(4)
used 3.2.2(3/2), 3.6(4), 3.7(5/5), 3.9.4(3/2), 3.10(6/2), 4.3.2(3), 4.4(3.2/4), 4.6(2), 4.7(2), 6.1(13/2), 6.1(15/5), 8.4(4/3), 8.5.1(2/5), 12.3(5), 12.4(2/3), 12.5(2.1/5), 12.5(2.2/5), 12.5.1(3/5), H.7.1(6/5), P.1
subtypes
of a profile 6.1(25)
subunit 1.3.3(37/5), 10.1.3(7), 10.1.3(8/2)
of a program unit 10.1.3(8/2)
used 10.1.1(3), P.1
Succ attribute 3.5(22)
Success
in Ada.Command_Line A.15(8)
successor element
of a hashed set A.18.8(68/2)
of a set A.18.7(6/2)
of an ordered set A.18.9(81/3)
successor node
of a hashed map A.18.5(46/2)
of a map A.18.4(6/2)
of an ordered map A.18.6(58/3)
Sunday
in Ada.Calendar.Formatting 9.6.1(17/2)
super
See view conversion 4.6(5/5)
Superscript_One
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Three
in Ada.Characters.Latin_1 A.3.3(22)
Superscript_Two
in Ada.Characters.Latin_1 A.3.3(22)
support external streaming 13.13.2(52/3)
Supported
in Ada.Execution_Time.Interrupts D.14.3(3/5)
suppress a check 1.3.5(6/5)
Suppress pragma 11.5(4/2), J.10(3/2), L(36)
suppressed check 11.5(8/2)
Suspend_Until_True
in Ada.Synchronous_Task_Control D.10(4/5)

Suspend_Until_True_And_Set_Deadline
  in Ada.Synchronous_Task_Control.EDF D.10(5.2/5)
Suspension_Object
  in Ada.Synchronous_Task_Control D.10(4/5)
Swap
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(28/5)
  in Ada.Containers.Indefinite_Holders A.18.18(22.1/5)
  in Ada.Containers.Multiway_Trees A.18.10(37/5)
  in Ada.Containers.Vectors A.18.2(55/5), A.18.2(56/5)
Swap_Links
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(29/5)
Symmetric_Difference
  in Ada.Containers.Hashed_Sets A.18.8(35/5), A.18.8(36/5)
  in Ada.Containers.Ordered_Sets A.18.9(36/5), A.18.9(37/5)
SYN
  in Ada.Characters.Latin_1 A.3.3(6)
synchronization 9(1/5)
Synchronization aspect 9.5(12/3)
synchronization_kind 9.5(10/3)
synchronized 1.3.1(78/5)
synchronized interface 3.9.4(5/2)
synchronized object 9.10(18/5)
synchronized tagged type 3.9.4(6/2)
Synchronized_Queue_Interfaces
  child of Ada.Containers A.18.27(3/5)
Synchronous_Barrier
  in Ada.Synchronous_Barriers D.10.1(5/3)
Synchronous_Barriers
  child of Ada D.10.1(3/5)
Synchronous_Task_Control
  child of Ada D.10(3/5)
syntactic category 1.1.4(15)
syntax
  complete listing P.1(1/5)
  cross reference P.2(1/3)
  notation 1.1.4(3)
  under Syntax heading 1.1.2(25)
System 13.7(3/5)
System.Address_To_Access_-Conversions 13.7.2(2/5)
System.Atomic_Operations C.6.1(3/5)
System.Atomic_Operations.Exchange C.6.2(5/5)
System.Atomic_Operations.Integer_Arithmetic C.6.4(3/5)
System.Atomic_Operations.Modular_Arithmetic C.6.5(3/5)
System.Atomic_Operations.Test_And_Set C.6.3(3/5)
System.Machine_Code 13.8(7)
System.Multiprocessors D.16(3/5)
System.Multiprocessors.Dispatching_Domains D.16.1(3/5)
System.RPC E.5(3/5)
System.Storage_Elements 13.7.1(2/5)
System.Storage_Pools 13.11(5/5)
System.Storage_Pools.Subpools 13.11.4(3/5)
System_Assertion_Check 11.5(23.11/5)
System_Dispatching_Domain
  in System.Multiprocessors.Dispatching_Domains D.16.1(6/3)
System_Name
  in System 13.7(4)
systems programming C(1)

T

Tag
  in Ada.Tags 3.9(6/5)
Tag attribute 3.9(16), 3.9(18)
tag indeterminate 3.9.2(6/2)
tag of an object 3.9(3)
  class-wide object 3.9(22)
  object created by an allocator 3.9(21/5)
  preserved by type conversion and parameter passing 3.9(25)
  returned by a function 3.9(23), 3.9(24/2)
  stand-alone object, component, or aggregate 3.9(20)
Tag_Array
  in Ada.Tags 3.9(7.3/2)
Tag_Check 11.5(18)
  [*partial*] 3.9.2(16), 4.6(42), 4.6(52), 5.2(10), 6.5(8.2/5)
Tag_Error
  in Ada.Tags 3.9(8)
tagged incomplete view 3.10.1(2.1/4)
tagged type 1.3.1(80/5), 3.9(2/2)
  protected 3.9.4(6/2)
  synchronized 3.9.4(6/2)
  task 3.9.4(6/2)
Tags
  child of Ada 3.9(6/5)
Tail
  in Ada.Strings.Bounded A.4.4(72), A.4.4(73)
  in Ada.Strings.Fixed A.4.3(37), A.4.3(38)
  in Ada.Strings.Unbounded A.4.5(67), A.4.5(68)
tail (of a queue) D.2.1(5/2)
tamper with cursors
  [*partial*] A.18(2.1/5)
  of a list A.18.3(61/5)
  of a map A.18.4(7/5)
  of a set A.18.7(7/5)
  of a tree A.18.10(80/5)
  of a vector A.18.2(90/5)
tamper with elements
  [*partial*] A.18(2.1/5)
  of a holder A.18.18(29/5)
  of a list A.18.3(61/5)
  of a map A.18.4(7/5)
  of a set A.18.7(7/5)
  of a tree A.18.10(80/5)
  of a vector A.18.2(90/5)
tampering
  prohibited for a holder A.18.18(29/5)
  prohibited for a list A.18.3(61/5)
  prohibited for a map A.18.4(7/5)
  prohibited for a set A.18.7(7/5)
  prohibited for a tree A.18.10(80/5)
  prohibited for a vector A.18.2(90/5)
Tampering_With_Cursors_Prohibited
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(10.1/5)
  in Ada.Containers.Hashed_Maps A.18.5(7.1/5)
  in Ada.Containers.Hashed_Sets A.18.8(8.1/5)
  in Ada.Containers.Multiway_Trees A.18.10(15.1/5)
  in Ada.Containers.Ordered_Maps A.18.6(8.1/5)
  in Ada.Containers.Ordered_Sets A.18.9(9.1/5)
  in Ada.Containers.Vectors A.18.2(12.1/5)
Tampering_With_Elements_Prohibited
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(10.2/5)
  in Ada.Containers.Hashed_Maps A.18.5(7.2/5)
  in Ada.Containers.Multiway_Trees A.18.10(15.2/5)
  in Ada.Containers.Ordered_Maps A.18.6(8.2/5)
  in Ada.Containers.Vectors A.18.2(12.2/5)
Tampering_With_The_Element_Prohibited
  in Ada.Containers.Indefinite_Holders A.18.18(8.1/5)
Tan
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(4)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(5)
Tanh
  in Ada.Numerics.Generic_Complex_-Elementary_Functions G.1.2(6)
  in Ada.Numerics.Generic_Elementary_-Functions A.5.1(7)
target
  of an assignment operation 5.2(3)
  of an assignment_statement 5.2(3)

target name 5.2.1(1/5)
target object
of a requeue_statement 9.5(7)
of the name of an entry or a protected subprogram 9.5(2/3)
target statement
of a goto_statement 5.8(3)
target subtype
of a type_conversion 4.6(3)
target_name 5.2.1(2/5)
used 4.1(2/5), P.1
task 9(1/5)
activation 9.2(1)
completion 9.3(1)
dependence 9.3(1)
execution 9.2(1)
termination 9.3(1)
task declaration 9.1(1)
task dispatching D.2.1(4/5)
task dispatching point D.2.1(4/5)
[*partial*] D.2.3(8/2), D.2.4(9/3)
task dispatching policy D.2.2(6/2)
[*partial*] D.2.1(5/2)
EDF_Within_Priorities D.2.6(7/5)
FIFO_Within_Priorities D.2.3(2/2)
Non_Preemptive_FIFO_Within_-Priorities D.2.4(2/2)
Round_Robin_Within_Priorities D.2.5(2/2)
task interface 3.9.4(5/2)
task priority D.1(15/5)
task state
abnormal 9.8(4)
blocked 9(10/5)
callable 9.9(2)
held D.11(4/2)
inactive 9(10/5)
ready 9(10/5)
terminated 9(10/5)
task tagged type 3.9.4(6/2)
task type 1.3.1(82/5)
task unit 9(9)
Task_Array
in Ada.Execution_Time.Group_Budgets D.14.2(6/2)
Task_Attributes
child of Ada C.7.2(2/5)
task_body 9.1(6/3)
used 3.11(6), P.1
task_body_stub 10.1.3(5/3)
used 10.1.3(2), P.1
task_definition 9.1(4)
used 9.1(2/3), 9.1(3/5), P.1
Task_Dispatching_Policy pragma D.2.2(2), L(37)
Task_Id
in Ada.Task_Identification C.7.1(2/5)
Task_Identification
child of Ada C.7.1(2/5)
task_item 9.1(5/1)
used 9.1(4), P.1
Task_Termination
child of Ada C.7.3(2/5)
task_type_declaration 9.1(2/3)
used 3.2.1(3/3), P.1
Tasking_Check 11.5(23.2/5)
[*partial*] 9.2(5), 9.5.3(21)
Tasking_Error
raised by detection of a bounded error 13.11.2(13)
raised by failure of runtime check 9.2(5), 9.5.3(21), 11.1(4), 11.5(23.1/5)
in Standard A.1(46)
template 12(1)
for a formal package 12.7(4)
See generic unit 12(1)
term 4.4(5)
used 4.4(4), P.1
terminal interrupt
example 9.7.4(10)
terminate_alternative 9.7.1(7)
used 9.7.1(4), P.1
terminated
a task state 9(10/5)
Terminated attribute 9.9(3)
termination
of a partition E.1(7)
termination handler C.7.3(8/3)
fall-back C.7.3(9/2)
specific C.7.3(9/2)
Termination_Handler
in Ada.Task_Termination C.7.3(4/2)
Terminator_Error
in Interfaces.C B.3(40)
Test_And_Set
child of System.Atomic_Operations C.6.3(3/5)
Test_And_Set_Flag
in System.Atomic_Operations.Test_And_Set C.6.3(4/5)
tested type
of a membership test 4.5.2(3/3)
text of a program 2.2(1)
Text_Buffer_Count
in Ada.Strings.Text_Buffers A.4.12(4/5)
Text_Buffers
child of Ada.Strings A.4.12(3/5)
Text_IO
child of Ada A.10.1(2/5)
Text_Streams
child of Ada.Text_IO A.12.2(3/5)
child of Ada.Wide_Text_IO A.12.3(3/5)
child of Ada.Wide_Wide_Text_IO A.12.4(3/5)
Text_Truncated
in Ada.Strings.Text_Buffers.Bounded A.4.12(28/5)
thread
logical 9(1/5)
throw (an exception)
See raise 11(1/3)
Thursday
in Ada.Calendar.Formatting 9.6.1(17/2)
tick 2.1(15/6)
in Ada.Real_Time D.8(6)
in System 13.7(10)
tied
accessibility levels 3.10.2(5/5)
Tilde
in Ada.Characters.Latin_1 A.3.3(14)
Time
in Ada.Calendar 9.6(10/5)
in Ada.Real_Time D.8(4)
time base 9.6(6/3)
time limit
example 9.7.4(12)
time type 9.6(6/3)
Time-dependent Reset procedure
of the random number generator A.5.2(34)
time-out
example 9.7.4(12)
See asynchronous_select 9.7.4(12)
See selective_accept 9.7.1(1)
See timed_entry_call 9.7.2(1/5)
Time_Error
in Ada.Calendar 9.6(18)
Time_First
in Ada.Real_Time D.8(4)
Time_Last
in Ada.Real_Time D.8(4)
Time_Of
in Ada.Calendar 9.6(15)
in Ada.Calendar.Formatting 9.6.1(30/2), 9.6.1(31/2)
in Ada.Execution_Time D.14(9/2)
in Ada.Real_Time D.8(16)
Time_Of_Event
in Ada.Real_Time.Timing_Events D.15(6/2)
Time_Offset
in Ada.Calendar.Time_Zones 9.6.1(4/2)
Time_Remaining
in Ada.Execution_Time.Timers D.14.1(8/2)
Time_Span
in Ada.Real_Time D.8(5)
Time_Span_First
in Ada.Real_Time D.8(5)
Time_Span_Last
in Ada.Real_Time D.8(5)

Time_Span_Unit
in Ada.Real_Time D.8(5)
Time_Span_Zero
in Ada.Real_Time D.8(5)
Time_Unit
in Ada.Real_Time D.8(4)
Time_Zones
child of Ada.Calendar 9.6.1(2/5)
timed_entry_call 9.7.2(2)
used 9.7(2), P.1
Timer
in Ada.Execution_Time.Timers D.14.1(4/2)
timer interrupt
example 9.7.4(12)
Timer_Handler
in Ada.Execution_Time.Timers D.14.1(5/5)
Timer_Resource_Error
in Ada.Execution_Time.Timers D.14.1(9/2)
Timers
child of Ada.Execution_Time D.14.1(3/5)
times operator 4.5(1), 4.5.5(1)
timing
See delay_statement 9.6(1)
Timing_Event
in Ada.Real_Time.Timing_Events D.15(4/5)
Timing_Event_Handler
in Ada.Real_Time.Timing_Events D.15(4/5)
Timing_Events
child of Ada.Real_Time D.15(3/5)
To_Ada
in Interfaces.C B.3(22), B.3(26), B.3(28), B.3(32), B.3(37), B.3(39), B.3(39.10/2), B.3(39.13/2), B.3(39.17/2), B.3(39.19/2), B.3(39.4/2), B.3(39.8/2)
in Interfaces.COBOL B.4(17), B.4(19)
in Interfaces.Fortran B.5(13), B.5(14), B.5(16)
To_Address
in System.Address_To_Access_-Conversions 13.7.2(3/3)
in System.Storage_Elements 13.7.1(10/3)
To_Basic
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20.1/5), A.3.5(21.1/5)
To_Big_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(7/5), A.5.6(12/5), A.5.6(13/5)
To_Big_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(9/5), A.5.7(13/5), A.5.7(14.1/6), A.5.7(14/5)
To_Binary
in Interfaces.COBOL B.4(45), B.4(48)
To_Bounded_String
in Ada.Strings.Bounded A.4.4(11)
To_C
in Interfaces.C B.3(21), B.3(25), B.3(27), B.3(32), B.3(36), B.3(38), B.3(39.13/2), B.3(39.16/2), B.3(39.18/2), B.3(39.4/2), B.3(39.7/2), B.3(39.9/2)
To_Character
in Ada.Characters.Conversions A.3.4(5/2)
To_Chars_Ptr
in Interfaces.C.Strings B.3.1(8)
To_COBOL
in Interfaces.COBOL B.4(17), B.4(18)
To_Cursor
in Ada.Containers.Vectors A.18.2(25/5)
To_Decimal
in Interfaces.COBOL B.4(35), B.4(40), B.4(44), B.4(47)
To_Display
in Interfaces.COBOL B.4(36)
To_Domain
in Ada.Strings.Maps A.4.2(24)
in Ada.Strings.Wide_Maps A.4.7(24)
in Ada.Strings.Wide_Wide_Maps A.4.8(24/2)
To_Duration
in Ada.Real_Time D.8(13)
To_Fortran
in Interfaces.Fortran B.5(13), B.5(14), B.5(15)
To_Holder
in Ada.Containers.Indefinite_Holders A.18.18(9/5)
To_Index
in Ada.Containers.Vectors A.18.2(26.1/5), A.18.2(26/5)
To_Integer
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(11/5)
in System.Storage_Elements 13.7.1(10/3)
To_ISO_646
in Ada.Characters.Handling A.3.2(11), A.3.2(12)
To_Long_Binary
in Interfaces.COBOL B.4(48)
To_Lower
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Mapping
in Ada.Strings.Maps A.4.2(23)
in Ada.Strings.Wide_Maps A.4.7(23)
in Ada.Strings.Wide_Wide_Maps A.4.8(23/2)
To_Packed
in Interfaces.COBOL B.4(41)
To_Picture
in Ada.Text_IO.Editing F.3.3(6)
To_Pointer
in System.Address_To_Access_-Conversions 13.7.2(3/3)
To_Quotient_String
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(19/5)
To_Range
in Ada.Strings.Maps A.4.2(24)
in Ada.Strings.Wide_Maps A.4.7(25)
in Ada.Strings.Wide_Wide_Maps A.4.8(25/2)
To_Ranges
in Ada.Strings.Maps A.4.2(10)
in Ada.Strings.Wide_Maps A.4.7(10)
in Ada.Strings.Wide_Wide_Maps A.4.8(10/2)
To_Real
in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(10/5)
To_Sequence
in Ada.Strings.Maps A.4.2(19)
in Ada.Strings.Wide_Maps A.4.7(19)
in Ada.Strings.Wide_Wide_Maps A.4.8(19/2)
To_Set
in Ada.Containers.Hashed_Sets A.18.8(9/5)
in Ada.Containers.Ordered_Sets A.18.9(10/5)
in Ada.Strings.Maps A.4.2(8), A.4.2(9), A.4.2(17), A.4.2(18)
in Ada.Strings.Wide_Maps A.4.7(8), A.4.7(9), A.4.7(17), A.4.7(18)
in Ada.Strings.Wide_Wide_Maps A.4.8(8/2), A.4.8(9/2), A.4.8(17/2), A.4.8(18/2)
To_String
in Ada.Characters.Conversions A.3.4(5/2)
in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(14/5)

in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(15/5)
in Ada.Strings.Bounded A.4.4(12)
in Ada.Strings.Unbounded A.4.5(11)
To_Time_Span
in Ada.Real_Time D.8(13)
To_Unbounded_String
in Ada.Strings.Unbounded A.4.5(9), A.4.5(10)
To_Upper
in Ada.Characters.Handling A.3.2(6), A.3.2(7)
in Ada.Wide_Characters.Handling A.3.5(20/3), A.3.5(21/3)
To_Vector
in Ada.Containers.Vectors A.18.2(13/5), A.18.2(14/5)
To_Wide_Character
in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_String
in Ada.Characters.Conversions A.3.4(4/2), A.3.4(5/2)
To_Wide_Wide_Character
in Ada.Characters.Conversions A.3.4(4/2)
To_Wide_Wide_String
in Ada.Characters.Conversions A.3.4(4/2)
token
See lexical element 2.2(1)
Trailing_Nonseparate
in Interfaces.COBOL B.4(23)
Trailing_Separate
in Interfaces.COBOL B.4(23)
transfer of control 5.1(14/2)
Translate
in Ada.Strings.Bounded A.4.4(53), A.4.4(54), A.4.4(55/6), A.4.4(56/6)
in Ada.Strings.Fixed A.4.3(18), A.4.3(19), A.4.3(20/6), A.4.3(21/6)
in Ada.Strings.Unbounded A.4.5(48), A.4.5(49), A.4.5(50/6), A.4.5(51/6)
Translation_Error
in Ada.Strings A.4.1(5)
Transpose
in Ada.Numerics.Generic_Complex_Arrays G.3.2(34/2)
in Ada.Numerics.Generic_Real_Arrays G.3.1(17/2)
Tree
in Ada.Containers.Multiway_Trees A.18.10(8/5), A.18.10(70.2/5)
Tree_Iterator_Interfaces
in Ada.Containers.Multiway_Trees A.18.10(13/3), A.18.10(70.7/5)
triggering_alternative 9.7.4(3)
used 9.7.4(2), P.1
triggering_statement 9.7.4(4/2)
used 9.7.4(3), P.1
Trim
in Ada.Strings.Bounded A.4.4(67), A.4.4(68), A.4.4(69)
in Ada.Strings.Fixed A.4.3(31), A.4.3(32), A.4.3(33), A.4.3(34)
in Ada.Strings.Unbounded A.4.5(61), A.4.5(62), A.4.5(63), A.4.5(64)
Trim_End
in Ada.Strings A.4.1(6)
True 3.5.3(1)
Truncation
in Ada.Strings A.4.1(6)
Truncation attribute A.5.3(42), A.5.4(12/6)
Tuesday
in Ada.Calendar.Formatting 9.6.1(17/2)
two's complement
modular types 3.5.4(29)
type 1.3.1(84/5), 3.2(1)
abstract 3.9.3(1.2/2)
needs finalization 7.6(9.1/2)
of a subtype 3.2(8/5)
stable property function of 7.3.4(7/5)
synchronized tagged 3.9.4(6/2)
See also tag 3.9(3)
See also language-defined types
type conformance 6.3.1(15/2)
[*partial*] 3.4(17/2), 8.3(8), 8.3(26/6), 10.1.4(4/3)
required 3.11.1(5), 4.1.4(17/5), 8.6(26), 9.1(9.2/3), 9.1(9.5/3), 9.4(11.1/3), 9.4(11.4/3), 9.5.4(3/3), 12.4(5/2)
type conversion 4.6(1/3)
access 4.6(24.11/2), 4.6(24.22/6), 4.6(24.23/6), 4.6(47)
arbitrary order 1.1.4(18)
array 4.6(24.2/2), 4.6(36)
composite (non-array) 4.6(21/3), 4.6(40)
enumeration 4.6(21.1/2), 4.6(34)
numeric 4.6(24.1/2), 4.6(29)
unchecked 13.9(1)
See also qualified_expression 4.7(1)
type conversion, implicit
See implicit subtype conversion 4.6(1/3)
type extension 3.9(2/2), 3.9.1(1/2)
type invariant 7.3.2(2/3)
class-wide 7.3.2(3/5)
See also invariant
type of a discrete_range 3.6.1(4)
type of a range 3.5(4)
type parameter
See discriminant 3.7(1/2)
type profile
See profile, type conformant 6.3.1(15/2)
type property aspect definition 7.3.4(3/5)
type resolution rules 8.6(20/2)
if any type in a specified class of types is expected 8.6(21)
if expected type is specific 8.6(22)
if expected type is universal or class-wide 8.6(21)
type tag
See tag 3.9(3)
type-related
aspect 13.1(8.1/3), 13.1(8/5)
operational item 13.1(8.1/3)
representation item 13.1(8/5)
type_conversion 4.6(2)
used 4.1(2/5), P.1
See also unchecked type conversion 13.9(1)
type_declaration 3.2.1(2)
used 3.1(3/3), P.1
type_definition 3.2.1(4/2)
used 3.2.1(3/3), P.1
Type_Invariant aspect 7.3.2(2/3)
Type_Invariant'Class aspect 7.3.2(3/5)
Type_Set
in Ada.Text_IO A.10.1(7)
types
of a profile 6.1(29)

U

UC_A_Acute
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Grave
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Ring
in Ada.Characters.Latin_1 A.3.3(23)
UC_A_Tilde
in Ada.Characters.Latin_1 A.3.3(23)
UC_AE_Diphthong
in Ada.Characters.Latin_1 A.3.3(23)
UC_C_Cedilla
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Acute
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)
UC_E_Grave
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Acute
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Circumflex
in Ada.Characters.Latin_1 A.3.3(23)
UC_I_Diaeresis
in Ada.Characters.Latin_1 A.3.3(23)

UC_I_Grave
  in Ada.Characters.Latin_1 A.3.3(23)
UC_Icelandic_Eth
  in Ada.Characters.Latin_1 A.3.3(24)
UC_Icelandic_Thorn
  in Ada.Characters.Latin_1 A.3.3(24)
UC_N_Tilde
  in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Acute
  in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Circumflex
  in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Diaeresis
  in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Grave
  in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Oblique_Stroke
  in Ada.Characters.Latin_1 A.3.3(24)
UC_O_Tilde
  in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Acute
  in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Circumflex
  in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Diaeresis
  in Ada.Characters.Latin_1 A.3.3(24)
UC_U_Grave
  in Ada.Characters.Latin_1 A.3.3(24)
UC_Y_Acute
  in Ada.Characters.Latin_1 A.3.3(24)
UCHAR_MAX
  in Interfaces.C B.3(6)
ultimate ancestor
  of a type 3.4.1(10/2)
unary adding operator 4.5.4(1)
unary operator 4.5(9)
unary_adding_operator 4.5(5)
  used 4.4(4), P.1
Unbiased_Rounding attribute A.5.3(39), A.5.4(10/6)
Unbounded
  child of Ada.Streams.Storage 13.13.1(18/5)
  child of Ada.Strings A.4.5(3/5)
  child of Ada.Strings.Text_Buffers A.4.12(18/5)
  in Ada.Text_IO A.10.1(5)
Unbounded_IO
  child of Ada.Text_IO A.10.12(3/5)
Unbounded_Priority_Queues
  child of Ada.Containers A.18.30(2/5)
Unbounded_Slice
  in Ada.Strings.Unbounded A.4.5(22.1/2), A.4.5(22.2/2)
Unbounded_String
  in Ada.Strings.Unbounded A.4.5(4/5)
Unbounded_Synchronized_Queues
  child of Ada.Containers A.18.28(2/5)
unchecked storage deallocation 13.11.2(1)
unchecked type conversion 13.9(1)
unchecked union object B.3.3(6/3)
unchecked union subtype B.3.3(6/3)
unchecked union type B.3.3(6/3)
Unchecked_Access attribute 13.10(3), H.4(18)
  See also Access attribute 3.10.2(24/1)
Unchecked_Conversion
  child of Ada 13.9(3/5)
Unchecked_Deallocation
  child of Ada 13.11.2(3/5)
Unchecked_Union aspect B.3.3(3.2/3)
Unchecked_Union pragma J.15.6(2/3), L(37.2/3)
unconditionally evaluated
  subexpression 6.1.1(22.11/5)
unconstrained 3.2(9)
  object 3.3.1(9/2)
  object 6.4.1(16)
  subtype 3.2(9), 3.4(6), 3.5(7), 3.5.1(10), 3.5.4(9), 3.5.4(10), 3.5.7(11), 3.5.9(13), 3.5.9(16), 3.6(15/5), 3.6(16), 3.7(26), 3.9(15)
  subtype 3.10(14/3)
  subtype K.2(33)
unconstrained_array_definition 3.6(3)
  used 3.6(2), P.1
undefined result 11.6(5/5)
underline 2.1(15/6)
  used 2.4.1(3), 2.4.2(4), P.1
unevaluated
  determined to be 6.1.1(24.1/5)
Unicode 1.2(9/6)
Uniformly_Distributed *subtype of* Float
  in Ada.Numerics.Float_Random A.5.2(8/5)
uninitialized allocator 4.8(4)
uninitialized variables 13.9.1(2)
  [*partial*] 3.3.1(21/5)
union
  C B.3.3(1/3)
  in Ada.Containers.Hashed_Sets A.18.8(26/5), A.18.8(27/5)
  in Ada.Containers.Ordered_Sets A.18.9(27/5), A.18.9(28/5)
unit consistency E.3(6)
unit matrix
  complex matrix G.3.2(148/2)
  real matrix G.3.1(80/2)
unit vector
  complex vector G.3.2(90/2)
  real vector G.3.1(48/2)
Unit_Matrix
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(51/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(29/2)
Unit_Vector
  in Ada.Numerics.Generic_Complex_Arrays G.3.2(24/2)
  in Ada.Numerics.Generic_Real_Arrays G.3.1(14/2)
universal type 3.4.1(6/5)
universal_access
  [*partial*] 3.4.1(6/5), 4.2(8/5)
universal_fixed
  [*partial*] 3.4.1(6/5), 3.5.6(4)
universal_integer
  [*partial*] 3.4.1(6/5), 3.5.4(14), 3.5.4(30), 4.2(8/5)
universal_real
  [*partial*] 3.4.1(6/5), 3.5.6(4), 4.2(8/5)
unknown discriminants 3.7(26)
unknown_discriminant_part 3.7(3)
  used 3.7(2/2), P.1
Unknown_Zone_Error
  in Ada.Calendar.Time_Zones 9.6.1(5/2)
unmarshalling E.4(9)
unsigned
  in Interfaces.C B.3(9)
  in Interfaces.COBOL B.4(23)
unsigned type
  See modular type 3.5.4(1)
unsigned_char
  in Interfaces.C B.3(10)
Unsigned_Conversions
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(13/5)
unsigned_long
  in Interfaces.C B.3(9)
Unsigned_N B.2(8)
unsigned_short
  in Interfaces.C B.3(9)
unspecified 1.1.3(18)

[*partial*] 2.1(5/6), 3.9(4/2), 3.9(12.5/3), 4.5.2(13), 4.5.2(24.2/1), 4.5.5(21), 4.6(58.5/5), 6.1.1(34/5), 6.1.1(35/3), 6.2(11/3), 7.2(5), 7.6(17.4/3), 9.8(14), 9.10(1/5), 10.2(26), 11.1(6/5), 11.4.1(10.1/4), 11.5(27/2), 13.1(18), 13.7.2(5/2), 13.9.1(7), 13.11(20), 13.11(21.6/5), 13.13.2(36/2), A.1(1/3), A.5.1(34), A.5.2(28), A.5.2(34), A.5.3(41.3/2), A.5.4(11.2/6), A.7(6/6), A.10(8/5), A.10.7(8/3), A.10.7(12/3), A.10.7(17.3/2), A.10.7(19), A.14(1), A.18.2(231/3), A.18.2(252/2), A.18.2(83/2), A.18.3(145/3), A.18.3(157/2), A.18.3(55/2), A.18.4(3/2), A.18.4(80/2), A.18.5(43/2), A.18.5(44/2), A.18.5(45/2), A.18.5(46/2), A.18.6(56/3), A.18.6(57/2), A.18.7(3/2), A.18.7(101/2), A.18.7(87/2), A.18.7(88/2), A.18.8(65/2), A.18.8(66.1/3), A.18.8(66/2), A.18.8(67/2), A.18.8(68/2), A.18.8(86/2), A.18.8(87/2), A.18.9(114/2), A.18.9(79.1/3), A.18.9(79/3), A.18.9(80/2), A.18.10(227/3), A.18.10(72/3), A.18.26(5/3), A.18.26(9.4/3), A.18.26(9/3), D.2.2(6.1/2), D.8(19), E.3(5/1), G.1.1(40), G.1.2(33), G.1.2(48), H(4.1), H.2(1/5), K.2(136.4/2), K.2(136.8/6)
Unsuppress pragma 11.5(4.1/2), L(37.3/2)
update
  the value of an object 3.3(14)
  in Interfaces.C.Strings B.3.1(18), B.3.1(19)
Update_Element
  in Ada.Containers.Doubly_Linked_-Lists A.18.3(17/5)
  in Ada.Containers.Hashed_Maps A.18.5(17/5)
  in Ada.Containers.Indefinite_Holders A.18.18(15/5)
  in Ada.Containers.Multiway_Trees A.18.10(27/5)
  in Ada.Containers.Ordered_Maps A.18.6(16/5)
  in Ada.Containers.Vectors A.18.2(33/5), A.18.2(34/5)
Update_Element_Preserving_Key
  in Ada.Containers.Hashed_Sets A.18.8(58/5)
  in Ada.Containers.Ordered_Sets A.18.9(73/5)
Update_Error
  in Interfaces.C.Strings B.3.1(20)
upper bound
  of a range 3.5(4)
upper bound is calculated 3.6.1(9/6)
upper-case letter
  a category of Character A.3.2(26)
Upper_Case_Map
  in Ada.Strings.Maps.Constants A.4.6(5)
Upper_Set
  in Ada.Strings.Maps.Constants A.4.6(4)
US
  in Ada.Characters.Latin_1 A.3.3(6)
usable as configuration pragma 2.8(6/6)
usage name 3.1(10)
usage profile
  Jorvik D.13(6.1/5)
  No_Implementation_Extensions 13.12.1(12/3)
  Ravenscar D.13(5/3)
use-visible 8.3(4), 8.4(9)
use_clause 8.4(2)
  used 3.11(4/1), 10.1.2(3), 12.1(5), P.1
Use_Error
  in Ada.Direct_IO A.8.4(18)
  in Ada.Directories A.16(43/2)
  in Ada.IO_Exceptions A.13(4)
  in Ada.Sequential_IO A.8.1(15)
  in Ada.Streams.Stream_IO A.12.1(26)
  in Ada.Text_IO A.10.1(85/5)
Use_Formal aspect H.7.1(3/5)
use_package_clause 8.4(3)
  used 8.4(2), P.1
use_type_clause 8.4(4/3)
  used 8.4(2), P.1
used
  generic formal parameters H.7.1(2/5)
user-defined assignment 7.6(1)
user-defined heap management 13.11(1)
user-defined literal 4.2.1(8/5)
user-defined literal aspect 4.2.1(2/5)
user-defined operator 6.6(1)
user-defined storage management 13.11(1)
UTC_Time_Offset
  in Ada.Calendar.Time_Zones 9.6.1(6.1/5)
UTF-16 A.4.11(46/3)
UTF-8 A.4.11(46/3)
UTF_16_Wide_String *subtype of* Wide_String
  in Ada.Strings.UTF_Encoding A.4.11(7/3)
UTF_8_String *subtype of* String
  in Ada.Strings.UTF_Encoding A.4.11(6/3)
UTF_Encoding
  child of Ada.Strings A.4.11(3/5)
UTF_String *subtype of* String
  in Ada.Strings.UTF_Encoding A.4.11(5/3)

V

Val attribute 3.5.5(5)
Valid
  in Ada.Text_IO.Editing F.3.3(5), F.3.3(12)
  in Interfaces.COBOL B.4(33), B.4(38), B.4(43)
Valid attribute 13.9.2(3/4), H(6/5)
Valid_Big_Integer *subtype of* Big_Integer
  in Ada.Numerics.Big_Numbers.Big_Integers A.5.6(5/5)
Valid_Big_Real *subtype of* Big_Real
  in Ada.Numerics.Big_Numbers.Big_Reals A.5.7(5/5)
value
  representation 13.4(10.4/5), K.2(59.4/5)
  in Ada.Calendar.Formatting 9.6.1(36/2), 9.6.1(38/2)
  in Ada.Environment_Variables A.17(4.1/3), A.17(4/2)
  in Ada.Numerics.Discrete_Random A.5.2(26)
  in Ada.Numerics.Float_Random A.5.2(14)
  in Ada.Strings.Maps A.4.2(21)
  in Ada.Strings.Wide_Maps A.4.7(21)
  in Ada.Strings.Wide_Wide_Maps A.4.8(21/2)
  in Ada.Task_Attributes C.7.2(4)
  in Interfaces.C.Pointers B.3.2(6), B.3.2(7)
  in Interfaces.C.Strings B.3.1(13), B.3.1(14), B.3.1(15), B.3.1(16)
Value attribute 3.5(52)
value conversion 4.6(5/5)
value_sequence 4.5.10(3/5)
  used 4.5.10(2/5), P.1
values
  belonging to a subtype 3.2(8/5)
variable 3.3(13/3)
  required 5.2(5/2), 6.4.1(5), 12.4(7), 13.11(15/5), 13.11.3(4/3)
variable indexing 4.1.6(16/3)
variable object 3.3(13/3)
variable view 3.3(13/3)
Variable_Indexing aspect 4.1.6(3/5)
variadic
  C B.3(60.16/4)
variant 3.8.1(3)
  used 3.8.1(2), P.1
  See also tagged type 3.9(1)
variant_part 3.8.1(2)

used 3.8(4), P.1
Vector
in Ada.Containers.Vectors A.18.2(8/5), A.18.2(79.2/5)
vector container A.18.2(1/2)
Vector_Iterator_Interfaces
in Ada.Containers.Vectors A.18.2(11.3/5), A.18.2(79.7/5)
Vectors
child of Ada.Containers A.18.2(6/5)
version
of a compilation unit E.3(5/1)
Version attribute E.3(3)
vertical line 2.1(15/6)
Vertical_Line
in Ada.Characters.Latin_1 A.3.3(14)
view 3.1(7)
of a subtype (implied) 3.1(7.1/3)
of a type (implied) 3.1(7.1/3)
of an object (implied) 3.1(7.1/3)
view conversion 4.6(5/5)
view of an entity 1.3.1(86/5)
virtual function
See dispatching subprogram 3.9.2(1/5)
Virtual_Length
in Interfaces.C.Pointers B.3.2(13)
visibility
direct 8.3(2), 8.3(21)
immediate 8.3(4), 8.3(21)
use clause 8.3(4), 8.4(9)
visibility rules 8.3(1)
visible 8.3(2), 8.3(14)
aspect_specification 8.3(23.1/3)
attribute_definition_clause 8.3(23.1/3)
within a pragma in a context_clause 10.1.6(3)
within a pragma that appears at the place of a compilation unit 10.1.6(5)
within a use_clause in a context_clause 10.1.6(3)
within a with_clause 10.1.6(2/2)
within the parent_unit_name of a library unit 10.1.6(2/2)
within the parent_unit_name of a subunit 10.1.6(4)
visible part 8.2(5)
of a formal package 12.7(10/2)
of a generic unit 8.2(8)
of a package (other than a generic formal package) 7.1(6/2)
of a protected unit 9.4(11/2)
of a task unit 9.1(9)
of a view of a callable entity 8.2(6)
of a view of a composite type 8.2(7)
visibly of an access type H.4(23.6/5)
volatile C.6(8/3)
Volatile aspect C.6(6.4/3)
Volatile pragma J.15.8(3/3), L(38.1/3)
Volatile_Components aspect C.6(6.8/3)
Volatile_Components pragma J.15.8(6/3), L(39.1/3)
VT
in Ada.Characters.Latin_1 A.3.3(5)
VTS
in Ada.Characters.Latin_1 A.3.3(17)

W

Wait_For_Release
in Ada.Synchronous_Barriers D.10.1(6/5)
wchar_array
in Interfaces.C B.3(33/3)
wchar_t
in Interfaces.C B.3(30/1)
Wednesday
in Ada.Calendar.Formatting 9.6.1(17/2)
well-formed picture String
for edited output F.3.1(1/3)
Wide_Bounded
child of Ada.Strings A.4.7(1/3)
Wide_Bounded_IO
child of Ada.Wide_Text_IO A.11(4/3)
Wide_Character 3.5.2(3/6)
in Standard A.1(36.1/6)
Wide_Character_Mapping
in Ada.Strings.Wide_Maps A.4.7(20/5)
Wide_Character_Mapping_Function
in Ada.Strings.Wide_Maps A.4.7(26)
Wide_Character_Range
in Ada.Strings.Wide_Maps A.4.7(6)
Wide_Character_Ranges
in Ada.Strings.Wide_Maps A.4.7(7)
Wide_Character_Sequence *subtype of* Wide_String
in Ada.Strings.Wide_Maps A.4.7(16)
Wide_Character_Set
in Ada.Strings.Wide_Maps A.4.7(4/5)
in Ada.Strings.Wide_Maps.Wide_-Constants A.4.8(48/2)
Wide_Characters
child of Ada A.3.1(4/5)
Wide_Command_Line
child of Ada A.15.1(2/5)
Wide_Constants
child of Ada.Strings.Wide_Maps A.4.7(1/3), A.4.8(28/2)
Wide_Directories
child of Ada A.16.2(2/5)
Wide_Environment_Variables
child of Ada A.17.1(2/5)
Wide_Equal_Case_Insensitive
child of Ada.Strings A.4.7(1/3)
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Exception_Name
in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Expanded_Name
in Ada.Tags 3.9(7/2)
Wide_File_Names
in Ada.Direct_IO A.8.4(18.1/5)
in Ada.Sequential_IO A.8.1(15.1/5)
in Ada.Streams.Stream_IO A.12.1(26.1/5)
in Ada.Text_IO A.10.1(85.1/5)
Wide_Fixed
child of Ada.Strings A.4.7(1/3)
Wide_Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(21/5)
Wide_Get_UTF_16
in Ada.Strings.Text_Buffers.Unbounded A.4.12(24/5)
Wide_Hash
child of Ada.Strings A.4.7(1/3)
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Hash_Case_Insensitive
child of Ada.Strings A.4.7(1/3)
child of Ada.Strings.Wide_Bounded A.4.7(1/3)
child of Ada.Strings.Wide_Fixed A.4.7(1/3)
child of Ada.Strings.Wide_Unbounded A.4.7(1/3)
Wide_Image attribute 4.10(31/5), 4.10(39/5)
Wide_Maps
child of Ada.Strings A.4.7(3/5)
wide_nul
in Interfaces.C B.3(31/1)
Wide_Put
in Ada.Strings.Text_Buffers A.4.12(8/5)
Wide_Put_UTF_16
in Ada.Strings.Text_Buffers A.4.12(11/5)
Wide_Space
in Ada.Strings A.4.1(4/2)
Wide_String
in Standard A.1(41/3)
Wide_Strings
child of Ada.Strings.UTF_Encoding A.4.11(30/5)
Wide_Text_IO
child of Ada A.11(2/2)

Wide_Unbounded
child of Ada.Strings A.4.7(1/3)
Wide_Unbounded_IO
child of Ada.Wide_Text_IO A.11(5/3)
Wide_Value attribute 3.5(40)
Wide_Wide_Bounded
child of Ada.Strings A.4.8(1/3)
Wide_Wide_Bounded_IO
child of Ada.Wide_Wide_Text_IO A.11(4/3)
Wide_Wide_Character 3.5.2(4/6)
in Standard A.1(36.2/6)
Wide_Wide_Character_Mapping
in Ada.Strings.Wide_Wide_Maps A.4.8(20/5)
Wide_Wide_Character_Mapping_Function
in Ada.Strings.Wide_Wide_Maps A.4.8(26/2)
Wide_Wide_Character_Range
in Ada.Strings.Wide_Wide_Maps A.4.8(6/2)
Wide_Wide_Character_Ranges
in Ada.Strings.Wide_Wide_Maps A.4.8(7/2)
Wide_Wide_Character_Sequence *subtype of* Wide_Wide_String
in Ada.Strings.Wide_Wide_Maps A.4.8(16/2)
Wide_Wide_Character_Set
in Ada.Strings.Wide_Wide_Maps A.4.8(4/5)
Wide_Wide_Characters
child of Ada A.3.1(6/5)
Wide_Wide_Command_Line
child of Ada A.15.1(3/5)
Wide_Wide_Constants
child of Ada.Strings.Wide_Wide_Maps A.4.8(1/3)
Wide_Wide_Directories
child of Ada A.16.2(3/5)
Wide_Wide_Environment_Variables
child of Ada A.17.1(3/5)
Wide_Wide_Equal_Case_Insensitive
child of Ada.Strings A.4.8(1/3)
child of Ada.Strings.Wide_Wide_-Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_-Unbounded A.4.8(1/3)
Wide_Wide_Exception_Name
in Ada.Exceptions 11.4.1(2/5), 11.4.1(5/2)
Wide_Wide_Expanded_Name
in Ada.Tags 3.9(7/2)
Wide_Wide_File_Names
in Ada.Direct_IO A.8.4(18.8/5)
in Ada.Sequential_IO A.8.1(15.8/5)
in Ada.Streams.Stream_IO A.12.1(26.8/5)
in Ada.Text_IO A.10.1(85.8/5)
Wide_Wide_Fixed
child of Ada.Strings A.4.8(1/3)
Wide_Wide_Get
in Ada.Strings.Text_Buffers.Unbounded A.4.12(22/5)
Wide_Wide_Hash
child of Ada.Strings A.4.8(1/3)
child of Ada.Strings.Wide_Wide_-Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_-Unbounded A.4.8(1/3)
Wide_Wide_Hash_Case_Insensitive
child of Ada.Strings A.4.8(1/3)
child of Ada.Strings.Wide_Wide_-Bounded A.4.8(1/3)
child of Ada.Strings.Wide_Wide_Fixed A.4.8(1/3)
child of Ada.Strings.Wide_Wide_-Unbounded A.4.8(1/3)
Wide_Wide_Image attribute 4.10(30/5), 4.10(38/5)
Wide_Wide_Maps
child of Ada.Strings A.4.8(3/5)
Wide_Wide_Put
in Ada.Strings.Text_Buffers A.4.12(9/5)
Wide_Wide_Space
in Ada.Strings A.4.1(4/2)
Wide_Wide_String
in Standard A.1(42.1/3)
Wide_Wide_Strings
child of Ada.Strings.UTF_Encoding A.4.11(38/5)
Wide_Wide_Text_IO
child of Ada A.11(3/2)
Wide_Wide_Unbounded
child of Ada.Strings A.4.8(1/3)
Wide_Wide_Unbounded_IO
child of Ada.Wide_Wide_Text_IO A.11(5/3)
Wide_Wide_Value attribute 3.5(39.1/2)
Wide_Wide_Width attribute 3.5(37.1/5)
Wide_Width attribute 3.5(38/5)
Width attribute 3.5(39/5)
with_clause 10.1.2(4/2)
mentioned in 10.1.2(6/2)
named in 10.1.2(6/2)
used 10.1.2(3), P.1
within
immediately 8.1(13/6)
word 13.3(8)
Word_Size
in System 13.7(13)
Write
in Ada.Direct_IO A.8.4(13/5)
in Ada.Sequential_IO A.8.1(12/5)
in Ada.Storage_IO A.9(7)
in Ada.Streams 13.13.1(6)
in Ada.Streams.Storage.Bounded 13.13.1(29/5)
in Ada.Streams.Storage.Unbounded 13.13.1(21/5)
in Ada.Streams.Stream_IO A.12.1(18/5), A.12.1(19/5)
in System.RPC E.5(8)
Write aspect 13.13.2(38/5)
Write attribute 13.13.2(3), 13.13.2(11)
Write clause 13.3(7/2), 13.13.2(38/5)
Write'Class aspect 13.13.2(38/5)

X

xor operator 4.5(1), 4.5.1(2)

Y

Year
in Ada.Calendar 9.6(13)
in Ada.Calendar.Formatting 9.6.1(21/2)
Year_Number *subtype of* Integer
in Ada.Calendar 9.6(11/2)
Yen_Sign
in Ada.Characters.Latin_1 A.3.3(21/3)
Yield
in Ada.Dispatching D.2.1(1.3/5)
Yield aspect D.2.1(1.7/5)
Yield_To_Higher
in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)
Yield_To_Same_Or_Higher
in Ada.Dispatching.Non_Preemptive D.2.4(2.2/5)