

INTERNATIONAL STANDARD

ISO/IEC
8652

Second edition
1995-02-15

Information technology — Programming languages — Ada

Technologies de l'information — Langages de programmation — Ada



Reference number
ISO/IEC 8652:1995
AVA Documents
Document Sharing Hub

Contents

Foreword	x
Introduction.....	xi
1. General	1
1.1 Scope	1
1.1.1 Extent	1
1.1.2 Structure	2
1.1.3 Conformity of an Implementation with the Standard	4
1.1.4 Method of Description and Syntax Notation	5
1.1.5 Classification of Errors	7
1.2 Normative References	8
1.3 Definitions	8
2. Lexical Elements	9
2.1 Character Set	9
2.2 Lexical Elements, Separators, and Delimiters	10
2.3 Identifiers	11
2.4 Numeric Literals	12
2.4.1 Decimal Literals	12
2.4.2 Based Literals	12
2.5 Character Literals	13
2.6 String Literals	13
2.7 Comments	14
2.8 Pragmas	14
2.9 Reserved Words	17
3. Declarations and Types	19
3.1 Declarations	19
3.2 Types and Subtypes	20
3.2.1 Type Declarations	21
3.2.2 Subtype Declarations	23
3.2.3 Classification of Operations	24
3.3 Objects and Named Numbers	24
3.3.1 Object Declarations	26
3.3.2 Number Declarations	28
3.4 Derived Types and Classes	28
3.4.1 Derivation Classes	31

© ISO/IEC 1995

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case Postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

3.5 Scalar Types	32
3.5.1 Enumeration Types	36
3.5.2 Character Types	37
3.5.3 Boolean Types	38
3.5.4 Integer Types	38
3.5.5 Operations of Discrete Types	41
3.5.6 Real Types	42
3.5.7 Floating Point Types	43
3.5.8 Operations of Floating Point Types	44
3.5.9 Fixed Point Types	45
3.5.10 Operations of Fixed Point Types	47
3.6 Array Types	48
3.6.1 Index Constraints and Discrete Ranges	50
3.6.2 Operations of Array Types	51
3.6.3 String Types	52
3.7 Discriminants	52
3.7.1 Discriminant Constraints	55
3.7.2 Operations of Discriminated Types	56
3.8 Record Types	56
3.8.1 Variant Parts and Discrete Choices	59
3.9 Tagged Types and Type Extensions	60
3.9.1 Type Extensions	62
3.9.2 Dispatching Operations of Tagged Types	63
3.9.3 Abstract Types and Subprograms	65
3.10 Access Types	67
3.10.1 Incomplete Type Declarations	69
3.10.2 Operations of Access Types	70
3.11 Declarative Parts	73
3.11.1 Completions of Declarations	74
4. Names and Expressions	75
4.1 Names	75
4.1.1 Indexed Components	76
4.1.2 Slices	77
4.1.3 Selected Components	78
4.1.4 Attributes	79
4.2 Literals	80
4.3 Aggregates	81
4.3.1 Record Aggregates	82
4.3.2 Extension Aggregates	83
4.3.3 Array Aggregates	84
4.4 Expressions	87
4.5 Operators and Expression Evaluation	88
4.5.1 Logical Operators and Short-circuit Control Forms	89
4.5.2 Relational Operators and Membership Tests	90
4.5.3 Binary Adding Operators	93
4.5.4 Unary Adding Operators	94
4.5.5 Multiplying Operators	94
4.5.6 Highest Precedence Operators	96
4.6 Type Conversions	97
4.7 Qualified Expressions	101
4.8 Allocators	102
4.9 Static Expressions and Static Subtypes	103
4.9.1 Statically Matching Constraints and Subtypes	105

5. Statements	107
5.1 Simple and Compound Statements - Sequences of Statements	107
5.2 Assignment Statements	108
5.3 If Statements	110
5.4 Case Statements	110
5.5 Loop Statements	112
5.6 Block Statements	113
5.7 Exit Statements	114
5.8 Goto Statements	115
6. Subprograms	117
6.1 Subprogram Declarations	117
6.2 Formal Parameter Modes	119
6.3 Subprogram Bodies	120
6.3.1 Conformance Rules	121
6.3.2 Inline Expansion of Subprograms	122
6.4 Subprogram Calls	123
6.4.1 Parameter Associations	124
6.5 Return Statements	125
6.6 Overloading of Operators	127
7. Packages	129
7.1 Package Specifications and Declarations	129
7.2 Package Bodies	130
7.3 Private Types and Private Extensions	131
7.3.1 Private Operations	133
7.4 Deferred Constants	135
7.5 Limited Types	136
7.6 User-Defined Assignment and Finalization	137
7.6.1 Completion and Finalization	139
8. Visibility Rules	143
8.1 Declarative Region	143
8.2 Scope of Declarations	144
8.3 Visibility	145
8.4 Use Clauses	147
8.5 Renaming Declarations	148
8.5.1 Object Renaming Declarations	148
8.5.2 Exception Renaming Declarations	149
8.5.3 Package Renaming Declarations	149
8.5.4 Subprogram Renaming Declarations	150
8.5.5 Generic Renaming Declarations	151
8.6 The Context of Overload Resolution	151
9. Tasks and Synchronization	155
9.1 Task Units and Task Objects	155
9.2 Task Execution - Task Activation	157
9.3 Task Dependence - Termination of Tasks	158
9.4 Protected Units and Protected Objects	159
9.5 Intertask Communication	162
9.5.1 Protected Subprograms and Protected Actions	163
9.5.2 Entries and Accept Statements	164
9.5.3 Entry Calls	167
9.5.4 Requeue Statements	169
9.6 Delay Statements, Duration, and Time	171
9.7 Select Statements	173

9.7.1 Selective Accept	174
9.7.2 Timed Entry Calls	176
9.7.3 Conditional Entry Calls	176
9.7.4 Asynchronous Transfer of Control	177
9.8 Abort of a Task - Abort of a Sequence of Statements	178
9.9 Task and Entry Attributes	179
9.10 Shared Variables	180
9.11 Example of Tasking and Synchronization	181
10. Program Structure and Compilation Issues	183
10.1 Separate Compilation	183
10.1.1 Compilation Units - Library Units	183
10.1.2 Context Clauses - With Clauses	186
10.1.3 Subunits of Compilation Units	186
10.1.4 The Compilation Process	188
10.1.5 Pragmas and Program Units	189
10.1.6 Environment-Level Visibility Rules	190
10.2 Program Execution	191
10.2.1 Elaboration Control	193
11. Exceptions	195
11.1 Exception Declarations	195
11.2 Exception Handlers	195
11.3 Raise Statements	196
11.4 Exception Handling	197
11.4.1 The Package Exceptions	197
11.4.2 Example of Exception Handling	199
11.5 Suppressing Checks	200
11.6 Exceptions and Optimization	202
12. Generic Units	205
12.1 Generic Declarations	205
12.2 Generic Bodies	206
12.3 Generic Instantiation	207
12.4 Formal Objects	210
12.5 Formal Types	211
12.5.1 Formal Private and Derived Types	212
12.5.2 Formal Scalar Types	213
12.5.3 Formal Array Types	214
12.5.4 Formal Access Types	215
12.6 Formal Subprograms	215
12.7 Formal Packages	217
12.8 Example of a Generic Package	217
13. Representation Issues	219
13.1 Representation Items	219
13.2 Pragma Pack	221
13.3 Representation Attributes	222
13.4 Enumeration Representation Clauses	227
13.5 Record Layout	228
13.5.1 Record Representation Clauses	228
13.5.2 Storage Place Attributes	230
13.5.3 Bit Ordering	230
13.6 Change of Representation	231
13.7 The Package System	232
13.7.1 The Package System.Storage_Elements	234

13.7.2 The Package System.Address_To_Access_Conversions	234
13.8 Machine Code Insertions	235
13.9 Unchecked Type Conversions	236
13.9.1 Data Validity	237
13.9.2 The Valid Attribute	238
13.10 Unchecked Access Value Creation	238
13.11 Storage Management	239
13.11.1 The Max_Size_In_Storage_Elements Attribute	242
13.11.2 Unchecked Storage Deallocation	242
13.11.3 Pragma Controlled	243
13.12 Pragma Restrictions	243
13.13 Streams	244
13.13.1 The Package Streams	244
13.13.2 Stream-Oriented Attributes	245
13.14 Freezing Rules	247

ANNEXES

A. Predefined Language Environment	251
A.1 The Package Standard	252
A.2 The Package Ada	255
A.3 Character Handling	255
A.3.1 The Package Characters	256
A.3.2 The Package Characters.Handling	256
A.3.3 The Package Characters.Latin_1	258
A.4 String Handling	262
A.4.1 The Package Strings	263
A.4.2 The Package Strings.Maps	263
A.4.3 Fixed-Length String Handling	266
A.4.4 Bounded-Length String Handling	273
A.4.5 Unbounded-Length String Handling	278
A.4.6 String-Handling Sets and Mappings	283
A.4.7 Wide_String Handling	283
A.5 The Numerics Packages	285
A.5.1 Elementary Functions	286
A.5.2 Random Number Generation	289
A.5.3 Attributes of Floating Point Types	293
A.5.4 Attributes of Fixed Point Types	297
A.6 Input-Output	297
A.7 External Files and File Objects	298
A.8 Sequential and Direct Files	299
A.8.1 The Generic Package Sequential_IO	299
A.8.2 File Management	300
A.8.3 Sequential Input-Output Operations	302
A.8.4 The Generic Package Direct_IO	303
A.8.5 Direct Input-Output Operations	304
A.9 The Generic Package Storage_IO	305
A.10 Text Input-Output	305
A.10.1 The Package Text_IO	307
A.10.2 Text File Management	311
A.10.3 Default Input, Output, and Error Files	312
A.10.4 Specification of Line and Page Lengths	313

A.10.5 Operations on Columns, Lines, and Pages	314
A.10.6 Get and Put Procedures	317
A.10.7 Input-Output of Characters and Strings	318
A.10.8 Input-Output for Integer Types	320
A.10.9 Input-Output for Real Types	322
A.10.10 Input-Output for Enumeration Types	325
A.11 Wide Text Input-Output	326
A.12 Stream Input-Output	326
A.12.1 The Package Streams.Stream_IO	326
A.12.2 The Package Text_IO.Text_Streams	328
A.12.3 The Package Wide_Text_IO.Text_Streams	329
A.13 Exceptions in Input-Output	329
A.14 File Sharing	330
A.15 The Package Command_Line	331
B. Interface to Other Languages	333
B.1 Interfacing Pragmas	333
B.2 The Package Interfaces	336
B.3 Interfacing with C	337
B.3.1 The Package Interfaces.C.Strings	341
B.3.2 The Generic Package Interfaces.C.Pointers	344
B.4 Interfacing with COBOL	347
B.5 Interfacing with Fortran	353
C. Systems Programming	357
C.1 Access to Machine Operations	357
C.2 Required Representation Support	358
C.3 Interrupt Support	358
C.3.1 Protected Procedure Handlers	360
C.3.2 The Package Interrupts	362
C.4 Prelaboration Requirements	364
C.5 Pragma Discard_Names	365
C.6 Shared Variable Control	365
C.7 Task Identification and Attributes	367
C.7.1 The Package Task_Identification	367
C.7.2 The Package Task_Attributes	368
D. Real-Time Systems	371
D.1 Task Priorities	371
D.2 Priority Scheduling	373
D.2.1 The Task Dispatching Model	373
D.2.2 The Standard Task Dispatching Policy	375
D.3 Priority Ceiling Locking	376
D.4 Entry Queuing Policies	378
D.5 Dynamic Priorities	379
D.6 Preemptive Abort	380
D.7 Tasking Restrictions	381
D.8 Monotonic Time	382
D.9 Delay Accuracy	386
D.10 Synchronous Task Control	387
D.11 Asynchronous Task Control	387
D.12 Other Optimizations and Determinism Rules	388
E. Distributed Systems	391
E.1 Partitions	391
E.2 Categorization of Library Units	393

E.2.1 Shared Passive Library Units	394
E.2.2 Remote Types Library Units	394
E.2.3 Remote Call Interface Library Units	395
E.3 Consistency of a Distributed System	396
E.4 Remote Subprogram Calls	397
E.4.1 Pragma Asynchronous	398
E.4.2 Example of Use of a Remote Access-to-Class-Wide Type	399
E.5 Partition Communication Subsystem	401
F. Information Systems	403
F.1 Machine_Radix Attribute Definition Clause	403
F.2 The Package Decimal	404
F.3 Edited Output for Decimal Types	405
F.3.1 Picture String Formation	406
F.3.2 Edited Output Generation	409
F.3.3 The Package Text_IO Editing	414
F.3.4 The Package Wide_Text_IO Editing	417
G. Numerics	419
G.1 Complex Arithmetic	419
G.1.1 Complex Types	419
G.1.2 Complex Elementary Functions	424
G.1.3 Complex Input-Output	427
G.1.4 The Package Wide_Text_IO.Complex_IO	429
G.2 Numeric Performance Requirements	430
G.2.1 Model of Floating Point Arithmetic	430
G.2.2 Model-Oriented Attributes of Floating Point Types	431
G.2.3 Model of Fixed Point Arithmetic	432
G.2.4 Accuracy Requirements for the Elementary Functions	434
G.2.5 Performance Requirements for Random Number Generation	436
G.2.6 Accuracy Requirements for Complex Arithmetic	436
H. Safety and Security	439
H.1 Pragma Normalize_Scalars	439
H.2 Documentation of Implementation Decisions	440
H.3 Reviewable Object Code	440
H.3.1 Pragma Reviewable	440
H.3.2 Pragma Inspection_Point	441
H.4 Safety and Security Restrictions	442
J. Obsolescent Features	445
J.1 Renamings of Ada 83 Library Units	445
J.2 Allowed Replacements of Characters	445
J.3 Reduced Accuracy Subtypes	446
J.4 The Constrained Attribute	446
J.5 ASCII	447
J.6 Numeric_Error	447
J.7 At Clauses	447
J.7.1 Interrupt Entries	448
J.8 Mod Clauses	449
J.9 The Storage_Size Attribute	449
K. Language-Defined Attributes	451
L. Language-Defined Pragmas	465

M. Implementation-Defined Characteristics..... 467

N. Glossary..... 473

P. Syntax Summary..... 477

Index 501

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 8652 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information Technology*.

This second edition cancels and replaces the first edition (ISO 8652:1987), of which it constitutes a technical revision.

Annexes A to J form an integral part of this International Standard. Annexes K to P are for information only.

Introduction

This is the Ada Reference Manual.

Other available Ada documents include:

- Rationale for the Ada Programming Language — 1995 edition, which gives an introduction to the new features of Ada, and explains the rationale behind them. Programmers should read this first.
- Changes to Ada — 1987 to 1995. This document lists in detail the changes made to the 1987 edition of the standard.
- The Annotated Ada Reference Manual (AARM). The AARM contains all of the text in the RM95, 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

Ada was originally designed with three overriding concerns: program reliability and maintenance, programming as a human activity, and efficiency. This 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.

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.

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.

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.

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.

Language Summary

An Ada program is composed of one or more program units. Program units may 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 must be visible to other units, and a body, containing the implementation details, which need not be visible to other units. Most program units can be compiled separately.

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.

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 must name the library units it requires.

Program Units

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 may read data, update variables, or produce some output. It may 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.

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.

Subprogram and package units may 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.

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

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 must be true before the body of the entry is executed. A protected unit may define a single protected object or a protected type permitting the creation of several similar objects.

Declarations and Statements

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.

The declarative part associates names with declared entities. For example, a name may 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.

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

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.

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.

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

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

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 may 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.

Execution of a program unit may encounter error situations in which normal program execution cannot continue. For example, an arithmetic computation may exceed the maximum allowed value of a number, or an attempt may 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.

Data Types

Every object in the language has a type, which characterizes a set of values and a set of applicable operations. The main classes 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, and Wide_Character are predefined.

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.

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 and Wide_String are predefined.

Record, task, and protected types may 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.

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 may designate the same object, and components of one object may 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.

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.

From any type a new type may be defined by derivation. A type, together with its derivatives (both direct and indirect) form a derivation class. Class-wide operations may be defined that accept as a parameter an operand of any type in a derivation class. For record and private types, the derivatives may be extensions of the parent type. Types that support these object-oriented capabilities of class-wide operations and type extension must be 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.

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.

Other Facilities

Representation 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 must 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.

The predefined environment of the language provides for input-output and other capabilities (such as string manipulation and random number generation) 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. Other standard library packages are defined in annexes of the standard to support systems with specialized requirements.

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.

Language Changes

This International Standard replaces the first edition of 1987. In this edition, the following major language changes have been incorporated:

- Support for standard 8-bit and 16-bit character sets. See Section 2, 3.5.2, 3.6.3, A.1, A.3, and A.4.
- Object-oriented programming with run-time polymorphism. See the discussions of classes, derived types, tagged types, record extensions, and private extensions in clauses 3.4, 3.9, and 7.3. See also the new forms of generic formal parameters that are allowed by 12.5.1, “Formal Private and Derived Types” and 12.7, “Formal Packages”.
- Access types have been extended to allow an access value to designate a subprogram or an object declared by an object declaration (as opposed to just a heap-allocated object). See 3.10.
- Efficient data-oriented synchronization is provided via protected types. See Section 9.
- The library units of a library may be organized into a hierarchy of parent and child units. See Section 10.
- Additional support has been added for interfacing to other languages. See Annex B.
- The Specialized Needs Annexes have been added to provide specific support for 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, “Safety and Security”

Instructions for Comment Submission

Informal comments on this International Standard may be sent via e-mail to ada-comment@sei.cmu.edu. If appropriate, the Project Editor will initiate the defect correction procedure.

Comments should use the following format:

!topic *Title summarizing comment*
!reference RM95-*ss.ss(pp)*
!from *Author Name yy-mm-dd*
!keywords *keywords related to topic*
!discussion

text of discussion

where *ss.ss* is the section, 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.

Multiple comments per e-mail message are acceptable. Please use a descriptive “Subject” in your e-mail message.

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:

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

Formal requests for interpretations and for reporting defects in this International Standard may be made in accordance with the ISO/IEC JTC 1 Directives and the ISO/IEC JTC 1/SC 22 policy for interpretations. National Bodies may submit a Defect Report to ISO/IEC JTC 1/SC 22 for resolution under the JTC 1 procedures. A response will be provided and, if appropriate, a Technical Corrigendum will be issued in accordance with the procedures.

Information technology — Programming languages — Ada

Section 1: General

Ada is a programming language designed to support the construction of long-lived, highly reliable software systems. The language includes facilities to define packages of related types, objects, and operations. The packages may be parameterized and the types may be extended to support the construction of libraries of reusable, adaptable software components. The operations may be implemented as subprograms using conventional sequential control structures, or as entries that include synchronization of concurrent threads of control as part of their invocation. The language treats modularity in the physical sense as well, with a facility to support separate compilation.

The language includes a complete facility for the support of real-time, concurrent programming. Errors can be signaled as exceptions and handled explicitly. The language also covers systems programming; this requires precise control over the representation of data and access to system-dependent properties. Finally, a predefined environment of standard packages is provided, including facilities for, among others, input-output, string manipulation, numeric elementary functions, and random number generation.

1.1 Scope

This International Standard specifies the form and meaning of programs written in Ada. Its purpose is to promote the portability of Ada programs to a variety of data processing systems.

1.1.1 Extent

This International Standard specifies:

- The form of a program written in Ada;
- The effect of translating and executing such a program;

- The manner in which program units may be combined to form Ada programs;
- The language-defined library units that a conforming implementation is required to supply;
- The permissible variations within the standard, and the manner in which they are to be documented;
- Those violations of the standard that a conforming implementation is required to detect, and the effect of attempting to translate or execute a program containing such violations;
- Those violations of the standard that a conforming implementation is not required to detect.

This International Standard does not specify:

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

1.1.2 Structure

This International Standard contains thirteen sections, fourteen annexes, and an index.

The *core* of the Ada language consists of:

- Sections 1 through 13
- Annex A, “Predefined Language Environment”
- Annex B, “Interface to Other Languages”
- 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, “Safety and Security”

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.

1.1.1 Extent

2

- Each clause or subclause whose title starts with the word ‘Example’ or ‘Examples’.

All implementations shall conform to the core language. In addition, an implementation may conform separately to one or more Specialized Needs Annexes.

The following Annexes are informative:

- Annex K, ‘Language-Defined Attributes’
- Annex L, ‘Language-Defined Pragmas’
- Annex M, ‘Implementation-Defined Characteristics’
- Annex N, ‘Glossary’
- Annex P, ‘Syntax Summary’

Each section is divided into clauses and subclauses that have a common structure. Each section, 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.

Bounded (Run-Time) Errors

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

Erroneous Execution

Situations that result in erroneous execution (see 1.1.5).

Implementation Requirements

Additional requirements for conforming implementations.

Documentation Requirements

Documentation requirements for conforming implementations.

Metrics

Metrics that are specified for the time/space properties of the execution of certain language constructs.

Implementation Permissions

Additional permissions given to the implementer.

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.

NOTES

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

Examples

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

1.1.3 Conformity of an Implementation with the Standard

Implementation Requirements

A conforming implementation shall:

- Translate and correctly execute legal programs written in Ada, provided that they are not so large as to exceed the capacity of the implementation;
- 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);
- Identify all programs or program units that contain errors whose detection is required by this International Standard;
- Supply all language-defined library units required by this International Standard;
- Contain no variations except those explicitly permitted by this International Standard, or those that are impossible or impractical to avoid given the implementation's execution environment;
- Specify all such variations in the manner prescribed by this International Standard.

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*:

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

A conforming implementation of this International Standard 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 International Standard for the semantics of the given program.

An implementation that conforms to this Standard shall support each capability required by the core language as specified. In addition, an implementation that conforms to this Standard may conform to one or more Specialized Needs Annexes (or to none). Conformance to a Specialized Needs Annex means that each capability required by the Annex is provided as specified.

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

Documentation Requirements

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 Annex M.

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

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.

NOTES

2 The above requirements imply that an implementation conforming to this Standard may support some of the capabilities required by a Specialized Needs Annex without supporting all required capabilities.

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.

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.

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

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

`case_statement`

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

array

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

```
return_statement ::= return [expression];
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.

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

- A vertical line separates alternative items unless it occurs immediately after an opening curly bracket, in which case it stands for itself:

```
constraint ::= scalar_constraint | composite_constraint
discrete_choice_list ::= discrete_choice { | discrete_choice }
```

- 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.

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*.

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

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

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 run-time 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.

NOTES

3 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:

```
if_statement ::=
    if condition then
        sequence_of_statements
    (elseif condition then
        sequence_of_statements)
    [else
        sequence_of_statements]
    end if;
```

4 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

The language definition classifies errors into several different categories:

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

These errors correspond to any violation of a rule given in this International Standard, 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.

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.

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

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;

The language rules define certain kinds of errors that need not 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`.

- Erroneous execution.

In addition to bounded errors, the language rules define certain kinds of errors as leading to *erroneous execution*. Like bounded errors, the implementation need not 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.

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 International Standard; in particular, in the standard mode, all legal compilation_units shall be accepted.

Implementation Advice

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

1.2 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

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

ISO/IEC 1539:1991, *Information technology — Programming languages — FORTRAN*.

ISO 1989:1985, *Programming languages — COBOL*.

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

ISO/IEC 8859-1:1987, *Information processing — 8-bit single-byte coded character sets — Part 1: Latin alphabet No. 1*.

ISO/IEC 9899:1990, *Programming languages — C*.

ISO/IEC 10646-1:1993, *Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane*.

1.3 Definitions

Terms are defined throughout this International Standard, indicated by *italic* type. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard 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 in Annex N, "Glossary".

Section 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 section. Pragmas, which provide certain information for the compiler, are also described in this section.

2.1 Character Set

The only characters allowed outside of comments are the `graphic_characters` and `format_effectors`.

Syntax

```
character ::= graphic_character | format_effector | other_control_function
graphic_character ::= identifier_letter | digit | space_character | special_character
```

Static Semantics

The character repertoire for the text of an Ada program consists of the collection of characters called the Basic Multilingual Plane (BMP) of the ISO 10646 Universal Multiple-Octet Coded Character Set, plus a set of `format_effectors` and, in comments only, a set of `other_control_functions`; the coded representation for these characters is implementation defined (it need not be a representation defined within ISO-10646-1).

The description of the language definition in this International Standard uses the graphic symbols defined for Row 00: Basic Latin and Row 00: Latin-1 Supplement of the ISO 10646 BMP; these correspond to the graphic symbols of ISO 8859-1 (Latin-1); no graphic symbols are used in this International Standard for characters outside of Row 00 of the BMP. The actual set of graphic symbols used by an implementation for the visual representation of the text of an Ada program is not specified.

The categories of characters are defined as follows:

<code>identifier_letter</code>	<code>upper_case_identifier_letter</code> <code>lower_case_identifier_letter</code>
<code>upper_case_identifier_letter</code>	Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Capital Letter".
<code>lower_case_identifier_letter</code>	Any character of Row 00 of ISO 10646 BMP whose name begins "Latin Small Letter".
<code>digit</code>	One of the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.
<code>space_character</code>	The character of ISO 10646 BMP named "Space".
<code>special_character</code>	Any character of the ISO 10646 BMP that is not reserved for a control function, and is not the <code>space_character</code> , an <code>identifier_letter</code> , or a <code>digit</code> .
<code>format_effector</code>	The control functions of ISO 6429 called character tabulation (HT), line tabulation (VT), carriage return (CR), line feed (LF), and form feed (FF).
<code>other_control_function</code>	Any control function, other than a <code>format_effector</code> , that is allowed in a comment; the set of <code>other_control_functions</code> allowed in comments is implementation defined.

The following names are used when referring to certain `special_characters`:

symbol	name	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	[	left square bracket
,	comma	]	right square bracket
–	hyphen-minus, minus	{	left curly bracket
.	full stop, dot, point	}	right curly bracket
/	solidus, divide		

Implementation Permissions

In a nonstandard mode, the implementation may support a different character repertoire; in particular, the set of characters that are considered `identifier_letters` can be extended or changed to conform to local conventions.

NOTES

1 Every code position of ISO 10646 BMP that is not reserved for a control function is defined to be a `graphic_character` by this International Standard. This includes all code positions other than 0000 - 001F, 007F - 009F, and FFFE - FFFF.

2 The language does not specify the source representation of programs.

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.

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 character tabulation (HT) signifies at least one end of line.

In some cases an explicit *separator* is required to separate adjacent lexical elements. A separator is any of a space character, a format effector, or the end of a line, as follows:

- A space character is a separator except within a comment, a `string_literal`, or a `character_literal`.
- Character tabulation (HT) is a separator except within a comment.
- The end of a line is always a separator.

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`.

A *delimiter* is either one of the following special characters

& ' () * + , - . / : ; < = > |

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

=> .. ** := /= >= <= << >> ◇

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`.

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

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.

Syntax

identifier ::=
 identifier_letter {[underline] letter_or_digit}

letter_or_digit ::= identifier_letter | digit

An identifier shall not be a reserved word.

Static Semantics

All characters of an identifier are significant, including any underline character. Identifiers differing only in the use of corresponding upper and lower case letters are considered the same.

Implementation Permissions

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

Examples

Examples of identifiers:

Count X Get_Symbol Ethelyn Marion

Snobol_4 X1 Page_Count Store_Next_Item

2.4 Numeric Literals

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

numeric_literal ::= decimal_literal | based_literal

NOTES

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

2.4.1 Decimal Literals

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

Syntax

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

numeral ::= digit { [underline] digit }

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

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

Static Semantics

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.

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

Examples of decimal literals:

12 0 1E6 123_456 -- integer literals

12.0 0.0 0.456 3.14159_26 -- real literals

2.4.2 Based Literals

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

Syntax

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

base ::= numeral

based_numeral ::=
extended_digit { [underline] extended_digit }

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.

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.

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

Examples

Examples of based literals:

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

2.5 Character Literals

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

Syntax

character_literal ::= 'graphic_character'

NOTES

4 A *character_literal* is an enumeration literal of a character type. See 3.5.2.

Examples

Examples of character literals:

'A' '*' ''' ''

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).

Syntax

string_literal ::= "{string_element}"

string_element ::= "" | *non_quotation_mark_graphic_character*

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

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

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

NOTES

5 An end of line cannot appear in a *string_literal*.

Examples

Examples of string literals:

"Message of the day:"

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

"Characters such as \$, %, and } are allowed in string literals"

2.7 Comments

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

Syntax

comment ::= --{*non_end_of_line_character*}

A comment may appear on any line of a program.

Static Semantics

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

Examples of comments:

-- *the last sentence above echoes the Algol 68 report*

end; -- *processing of Line is complete*

-- *a long comment may be split onto*

-- *two or more consecutive lines*

----- *the first two hyphens start the comment*

2.8 Pragmas

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

pragma ::=
 pragma **identifier** [(**pragma_argument_association** { , **pragma_argument_association** })];
pragma_argument_association ::=
 [**pragma_argument_identifier** =>] **name**
 | [**pragma_argument_identifier** =>] **expression**

In a pragma, any pragma_argument_associations without a *pragma_argument_identifier* shall precede any associations with a *pragma_argument_identifier*.

Pragmas are only allowed at the following places in a program:

- After a semicolon delimiter, but not within a formal_part or discriminant_part.
- At any place where the syntax rules allow a construct defined by a syntactic category whose name ends with "declaration", "statement", "clause", or "alternative", or one of the syntactic categories variant or exception_handler; but not in place of such a construct. Also at any place where a compilation_unit would be allowed.

Additional syntax rules and placement restrictions exist for specific pragmas.

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*.

An *identifier specific to a pragma* is an identifier that is used in a pragma argument with special meaning for that pragma.

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.

Dynamic Semantics

Any pragma that appears at the place of an executable construct is executed. Unless otherwise specified for a particular pragma, this execution consists of the evaluation of each evaluable pragma argument in an arbitrary order.

Implementation Requirements

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

Implementation Permissions

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

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

Normally, implementation-defined pragmas should have no semantic effect for error-free programs; that is, if the implementation-defined pragmas are removed from a working program, the program should still be legal, and should still have the same semantics.

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

- A pragma used to complete a declaration, such as a pragma Import;
- A pragma used to configure the environment by adding, removing, or replacing library_items.

Syntax

The forms of List, Page, and Optimize pragmas are as follows:

pragma List(identifier);

pragma Page;

pragma Optimize(identifier);

Other pragmas are defined throughout this International Standard, and are summarized in Annex L.

Static Semantics

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.

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).

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

Examples of pragmas:

```
pragma List(Off); -- turn off listing generation
pragma Optimize(Off); -- turn off optional optimizations
pragma Inline(Set_Mask); -- generate code for Set_Mask inline
pragma Suppress(Range_Check, On => Index); -- turn off range checking on Index
```

2.9 Reserved Words

Syntax

The following are the *reserved words* (ignoring upper/lower case distinctions):

abort	else	new	return
abs	elsif	not	reverse
abstract	end	null	
accept	entry		select
access	exception		separate
aliased	exit	of	subtype
all		or	
and	for	others	tagged
array	function	out	task
at			terminate
	generic	package	then
begin	goto	pragma	type
body		private	
	if	procedure	
case	in	protected	until
constant	is		use
		raise	
declare		range	when
delay	limited	record	while
delta	loop	rem	with
digits		renames	
do	mod	requeue	xor

NOTES

6 The reserved words appear in **lower case boldface** in this International Standard, 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 may be written in whatever typeface is desired and available.

Section 3: Declarations and Types

This section describes the types in the language and the rules for declaring constants, variables, and named numbers.

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 several forms of declaration. A *basic_declaration* is a form of declaration defined as follows.

Syntax

```
basic_declaration ::=
    type_declaration           | subtype_declaration
  | object_declaration         | number_declaration
  | subprogram_declaration     | abstract_subprogram_declaration
  | package_declaration       | renaming_declaration
  | exception_declaration     | generic_declaration
  | generic_instantiation
defining_identifier ::= identifier
```

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).

Each of the following is defined to be a declaration: any *basic_declaration*; an *enumeration_literal_specification*; a *discriminant_specification*; a *component_declaration*; a *loop_parameter_specification*; a *parameter_specification*; a *subprogram_body*; an *entry_declaration*; an *entry_index_specification*; a *choice_parameter_specification*; a *generic_formal_parameter_declaration*.

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)).

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.

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).

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

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 International Standard for each construct that has a run-time effect.

NOTES

1 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

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.

Types are grouped into *classes* of types, reflecting the similarity of their values and primitive operations. There exist several *language-defined classes* of types (see NOTES below). *Elementary* types are those whose values are logically indivisible; *composite* types are those whose values are composed of *component* values.

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*.

The composite types are the *record types*, *record extensions*, *array types*, *task types*, and *protected types*. A *private type* or *private extension* represents a partial view (see 7.3) of a type, providing support for data abstraction. A partial view is a composite type.

Certain composite types (and partial 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.

The term *subcomponent* is used in this International Standard 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 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*.

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. Such values *belong* to the subtype.

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).

NOTES

2 Any set of types that is closed under derivation (see 3.4) can be called a “class” of types. However, only certain 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.

These language-defined classes are organized like this:

```

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
    array
      string
      other array
    untagged record
    tagged
    task
    protected

```

The classes “numeric” and “nonlimited” represent other classification dimensions and do not fit into the above strictly hierarchical picture.

3.2.1 Type Declarations

A `type_declaration` declares a type and its first subtype.

Syntax

```

type_declaration ::= full_type_declaration
                  | incomplete_type_declaration
                  | private_type_declaration
                  | private_extension_declaration

full_type_declaration ::=
    type_defining_identifier [known_discriminant_part] is type_definition;
    | task_type_declaration
    | protected_type_declaration

```

```

type_definition ::=
    enumeration_type_definition | integer_type_definition
    | real_type_definition       | array_type_definition
    | record_type_definition     | access_type_definition
    | derived_type_definition

```

Legality Rules

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

Static Semantics

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, “Discriminants”). The remainder of the *type_declaration* defines the remaining characteristics of (the view of) the type.

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 (including a parameter or a discriminant). The type defined by such a declaration is *anonymous* — it has no nameable subtypes. For explanatory purposes, this International Standard 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 International Standard 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 as part of declaring an object of the type, is called a *full 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.

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, “Operators and Expression Evaluation”.

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.

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.

*Examples**Examples of type definitions:*

```

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

```

Examples of type declarations:

```

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

```

NOTES

3 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 International Standard 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.

3.2.2 Subtype Declarations

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

Syntax

```

subtype_declaration ::=
    subtype_defining_identifier is subtype_indication;
subtype_indication ::= subtype_mark [constraint]
subtype_mark ::= subtype_name
constraint ::= scalar_constraint | composite_constraint
scalar_constraint ::=
    range_constraint | digits_constraint | delta_constraint
composite_constraint ::=
    index_constraint | discriminant_constraint

```

Name Resolution Rules

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

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:

- The constraint is first elaborated.
- A check is then made that the constraint is *compatible* with the subtype denoted by the `subtype_mark`.

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.

NOTES

4 A `scalar_constraint` may 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` may 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

Examples of subtype declarations:

```

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

```

3.2.3 Classification of Operations

Static Semantics

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 (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.

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

- The predefined operators of the type (see 4.5);
- For a derived type, the inherited (see 3.4) user-defined subprograms;
- For an enumeration type, the enumeration literals (which are considered parameterless functions — see 3.5.1);
- 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;
- 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.

A primitive subprogram whose designator is an `operator_symbol` is called a *primitive operator*.

3.3 Objects and Named Numbers

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

All of the following are objects:

- the entity declared by an `object_declaration`;
- a formal parameter of a subprogram, entry, or generic subprogram;
- a generic formal object;
- a loop parameter;
- a choice parameter of an `exception_handler`;
- an entry index of an `entry_body`;
- the result of dereferencing an access-to-object value (see 4.1);

3.2.2 Subtype Declarations

- the result of evaluating a `function_call` (or the equivalent operator invocation — see 6.6);
- the result of evaluating an aggregate;
- a component, slice, or view conversion of another object.

An object is either a *constant* object or a *variable* object. The value of a constant object cannot be changed between its initialization and its finalization, whereas the value of a variable object can be changed. Similarly, a view of an object is either a *constant* or a *variable*. All views of a constant object are constant. A constant view of a variable object cannot be used to modify the value of the variable. The terms constant and variable by themselves refer to constant and variable views of objects.

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.

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

- an object declared by an `object_declaration` with the reserved word **constant**;
- a formal parameter or generic formal object of mode **in**;
- a discriminant;
- a loop parameter, choice parameter, or entry index;
- the dereference of an access-to-constant value;
- the result of evaluating a `function_call` or an aggregate;
- a `selected_component`, `indexed_component`, slice, or view conversion of a constant.

At the place where a view of an object is defined, a *nominal subtype* is associated with 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 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.

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

NOTES

5 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.

6 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, task, or protected object, the `object_declaration` may include the definition of the (anonymous) type of the object.

Syntax

```
object_declaration ::=
    defining_identifier_list : [ aliased ] [ constant ] subtype_indication [ := expression ];
    | defining_identifier_list : [ aliased ] [ constant ] array_type_definition [ := expression ];
    | single_task_declaration
    | single_protected_declaration
defining_identifier_list ::=
    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*.

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. An initialization expression shall not be given if the object is of a limited type.

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 clause 7.4. The rules for full constant declarations are given in this subclause.

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 International Standard relies on this equivalence; explanations are given for declarations with a single `defining_identifier`.

The `subtype_indication` 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.

Dynamic Semantics

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 or aliased (see 3.10) 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*. In the case of an aliased object, this initial value may be either explicit or implicit; in the other cases, an explicit initial value is required. 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*.

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:

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

- The implicit initial (and only) value for each discriminant of a constrained discriminated subtype is defined by the subtype.
- 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 might raise `Constraint_Error` — see 4.6, “Type Conversions”), 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`, any implicit initial values are those determined by the component's nominal subtype.
- 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.

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

1. The `subtype_indication`, `array_type_definition`, `single_task_declaration`, or `single_protected_declaration` is first elaborated. This creates the nominal subtype (and the anonymous type in the latter three cases).
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 might raise `Constraint_Error` — see 4.6).
3. The object is created, and, if there is not an initialization expression, any per-object expressions (see 3.8) are evaluated and any implicit initial values for the object or for its subcomponents are obtained as determined by the nominal subtype.
4. 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.

For the third step above, the object creation and any elaborations and evaluations are performed in an arbitrary order, except that if the `default_expression` for a discriminant is evaluated to obtain its initial value, then this evaluation is performed before that of the `default_expression` for any component that depends on the discriminant, and also before that of any `default_expression` that includes the name of the discriminant. The evaluations of the third step and the assignments of the fourth step are performed in an arbitrary order, except that each evaluation is performed before the resulting value is assigned.

There is no implicit initial value defined for a scalar subtype. In the absence of an explicit initialization, a newly created scalar object might have a value that does not belong to its subtype (see 13.9.1 and H.1).

NOTES

7 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.

8 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`, `parameter_specification`, `entry_index_specification`, `choice_parameter_specification`, or a `formal_object_declaration` is not called a stand-alone object.

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

Examples

Example of a multiple object declaration:

-- the multiple object declaration

John, Paul : `Person_Name` := `new` `Person`(Sex => M); -- see 3.10.1

-- is equivalent to the two single object declarations in the order given

```
John : Person_Name := new Person(Sex => M);
Paul : Person_Name := new Person(Sex => M);
```

Examples of variable declarations:

```
Count, Sum : Integer;
Size       : Integer range 0 .. 10_000 := 0;
Sorted     : Boolean := False;
Color_Table : array(1 .. Max) of Color;
Option     : Bit_Vector(1 .. 10) := (others => True);
Hello      : constant String := "Hi, world.";
```

Examples of constant declarations:

```
Limit      : constant Integer := 10_000;
Low_Limit  : constant Integer := Limit/10;
Tolerance  : constant Real := Dispersion(1.15);
```

3.3.2 Number Declarations

A *number_declaration* declares a named number.

Syntax

```
number_declaration ::=
  defining_identifier_list : constant := static_expression;
```

Name Resolution Rules

The *static_expression* given for a *number_declaration* is expected to be of any numeric type.

Legality Rules

The *static_expression* given for a number declaration shall be a static expression, as defined by clause 4.9.

Static Semantics

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.

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

Dynamic Semantics

The elaboration of a *number_declaration* has no effect.

Examples

Examples of number declarations:

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

A *derived_type_definition* defines a new type (and its first subtype) whose characteristics are *derived* from those of a *parent type*.

Syntax

`derived_type_definition ::= [abstract] new parent_subtype_indication [record_extension_part]`

Legality Rules

The *parent_subtype_indication* defines the *parent subtype*; its type is the parent type.

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*.

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.

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.

The characteristics of the derived type are defined as follows:

- Each class of types that includes the parent type also includes the derived type.
- 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.
- 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:
 - 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;
 - 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*;
 - Each component declared in a *record_extension_part*, if any.

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*.

- The derived type is limited if and only if the parent type is limited.
- For each predefined operator of the parent type, there is a corresponding predefined operator of the derived type.
- For each user-defined primitive subprogram (other than a user-defined equality operator — see below) of the parent 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 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).

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 type, after systematic replacement of each subtype of its profile (see 6.1) that is of the parent type with a *corresponding subtype* of the derived type. For a given subtype of the parent type, the corresponding subtype of the derived type is defined as follows:

- 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.
- If the derived type is a record extension, then the corresponding subtype is the first subtype of the derived type.
- 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).

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 type by the derived type is handled as part of the normal type conversion associated with parameter passing — see 6.4.1.

If a primitive subprogram of the parent 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.

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.

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.

For the execution of a call on an inherited subprogram, a call on the corresponding primitive subprogram of the parent 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 parent's subprogram is converted to the derived type.

NOTES

10 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.

11 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.

12 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.

- 13 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.
- 14 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).
- 15 Any discriminants of the parent type are either all inherited, or completely replaced with a new set of discriminants.
- 16 For an inherited subprogram, the subtype of a formal parameter of the derived type need not have any value in common with the first subtype of the derived type.
- 17 If the reserved word **abstract** is given in the declaration of a type, the type is abstract (see 3.9.3).

Examples

Examples of derived type declarations:

```

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
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;
```

3.4.1 Derivation Classes

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

Static Semantics

A derived type is *derived from* its parent type *directly*; it is derived *indirectly* from any type from which its parent type is derived. 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).

Every type is either 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 a declaration for an object. Class-wide and universal types are implicitly defined, to act as representatives for an entire class of types, as follows:

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).

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.

Universal types Universal types are defined for (and belong to) the integer, real, and fixed point classes, and are referred to in this standard as respectively, *universal_integer*, *universal_real*, and *universal_fixed*. These are analogous to class-wide types for these language-defined numeric classes. As with class-wide types, if a formal parameter is of a universal type, then an actual parameter of any type in the cor-

responding class is acceptable. In addition, a value of a universal type (including an integer or real numeric_literal) is “universal” in that it is acceptable where some particular type in the class is expected (see 8.6).

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*.

A class-wide or universal type is said to *cover* all of the types in its class. A specific type covers only itself.

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* is defined to be a descendant of type *T1* if *T2* is a descendant of *T1*. Similarly, the 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*. The *ultimate ancestor* of a type is the ancestor of the type that is not a descendant of any other type.

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.

NOTES

18 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

$$1 + 4 < 7$$

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.

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.

Syntax

```
range_constraint ::= range range
range ::= range_attribute_reference
| 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.

3.4.1 Derivation Classes

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

Static Semantics

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.

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

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.

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.

Attributes

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

S'First	S'First denotes the lower bound of the range of S. The value of this attribute is of the type of S.
S'Last	S'Last denotes the upper bound of the range of S. The value of this attribute is of the type of S.
S'Range	S'Range is equivalent to the range S'First .. S'Last.
S'Base	S'Base denotes an unconstrained subtype of the type of S. This unconstrained subtype is called the <i>base subtype</i> of the type.
S'Min	S'Min denotes a function with the following specification: <pre> function S'Min(Left, Right : S'Base) return S'Base </pre> The function returns the lesser of the values of the two parameters.
S'Max	S'Max denotes a function with the following specification: <pre> function S'Max(Left, Right : S'Base) return S'Base </pre> The function returns the greater of the values of the two parameters.
S'Succ	S'Succ denotes a function with the following specification: <pre> function S'Succ(Arg : S'Base) return S'Base </pre>

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:

```
function S'Pred(Arg : S'Base)
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.

S'Wide_Image

S'Wide_Image denotes a function with the following specification:

```
function S'Wide_Image(Arg : S'Base)
return Wide_String
```

The function returns an *image* of the value of *Arg*, that is, a sequence of characters representing the value in display form. The lower bound of the result is one.

The image of an integer value 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.

The image of an enumeration value 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 result is a corresponding language-defined or implementation-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).

The image of a floating point value 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.

The image of a fixed point value 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, 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.

S'Image

S'Image denotes a function with the following specification:

```
function S'Image(Arg : S'Base)
return String
```

The function returns an image of the value of *Arg* as a String. The lower bound of the result is one. The image has the same sequence of graphic characters as that defined for S'Wide_Image if all the graphic characters are defined in Character; otherwise the sequence of characters is implementation defined (but no shorter than that of S'Wide_Image for the same value of *Arg*).