acm

SIGPLAN NOTICES

A Monthly Publication of
the Special Interest Group
on Programming Languages

VOLUME 14, NUMBER 6, JUNE 1979 PART A

Contents:

PRELIMINARY ADA REFERENCE MANUAL

SIGPLAN NOTICES

special interest group on programming languages

SIGPLAN *Notices* is an informal monthly publication of the Special Interest Group on Programming Languages (SIGPLAN) of the Association for Computing Machinery.

Membership in SIGPLAN is open to ACM Members or associate members for $11 per year. It is also open to others whose major professional allegiance is in a field other than information processing or computing for $19 per year. All SIGPLAN members receive SIGPLAN *Notices*, are given discounts at SIGPLAN-sponsored meetings, and may vote in the Group's biennial elections. ACM members of SIGPLAN may serve as officers of the group.

Institutional or Library subscriptions to SIGPLAN *Notices* are available for $25 per year, and the regular back issues of the *Notices* may be purchased for $3 per copy from ACM Headquarters. SIGPLAN symposium and conference proceedings issues of the *Notices* are available, at different prices, from ACM Headquarters.

Members of the Special Technical Committee on APL (STAPL), a SIGPLAN-affiliated users' group, receive the *APL Quote-Quad*, an informal quarterly publication of interest to APL users.

All correspondence concerning STAPL should be directed to the committee chairman, Philip S. Abrams, Scientific Time Sharing Corp., 7316 Wisconsin Avenue, Bethesda, Maryland 20014.

CONTRIBUTIONS TO SIGPLAN *Notices* may be sent to the Editor. They should be camera-ready and typed with single spacing. Contributed papers 15 pages or longer will be returned to authors for retyping on model paper and ultimate photoreduction. Announcements or copies of relevant company reports and academic theses are requested for inclusion in the Publications section. Those interested in editing a special issue devoted to a specific topic are invited to contact the Editor to discuss the matter. Letters to the Editor of SIGPLAN *Notices* will be considered as submitted for publication unless they contain a request to the contrary.

Technical papers appearing in this issue are unrefereed working papers, and all contributions are ordinarily to be construed as personal rather than organizational statements. Authors planning to submit to a journal any manuscript appearing in SIGPLAN *Notices* should make certain that the journal version differs sufficiently in content to satisfy the editor of that journal. All questions regarding journal policy on possible duplicate publications should be directed to the editor of the journal in question.

PROSPECTIVE ORGANIZERS OF SIGPLAN TECHNICAL MEETINGS are referred to the Volume 8, No. 4, (April 1973) issue of SIGPLAN *Notices* in which the "Suggested Guidelines for SIGPLAN Technical Meetings were published.

CHANGES OF ADDRESS AND QUESTIONS PERTAINING TO THE MAILING OF SIGPLAN *Notices* should be sent to ACM Headquarters:

ACM SIGPLAN, 1133 Avenue of the Americas,
New York, New York 10036
Phone: 212/265-6300

EXECUTIVE COMMITTEE

Chairman
Stephen N. Zilles
IBM Research Lab
K-54/282
Monterey & Cottle Roads
San Jose, CA 95193
(408) 256-7559

Vice-Chairman
Paul Abrahams
Courant Institute
251 Mercer Street
New York, NY 10012
(212) 460-7387

Secretary/Treasurer
John R. White
Electrical Engineering
& Computer Science
University of Connecticut
Storrs, CT 06268
(203) 486-2572/4816

Members

Allen Ambler
Amdahl Corporation
1250 East Arques Avenue
Sunnyvale, CA 94086
(408) 746-6567

Anita K. Jones
Carnegie-Mellon University
Computer Science Department
Schenley Park
Pittsburgh, PA 15213
(412) 363-8666

Barbara Liskov
Project MAC—Rm 528
M.I.T.
Cambridge, MA 02139
(617) 253-5886

Editor

Victor B. Schneider
The Aerospace Corporation
Post Office Box 92957
Los Angeles, CA 90009
(213) 648-5405

Past Chairman

Robert M. Graham
Department of Computer
and Information Science
Graduate Research Center
University of Massachusetts
Amherst, MA 01002
(413) 545-2744

STAPL Chairman

Philip S. Abrams
Scientific Time Sharing Corp.
7316 Wisconsin Avenue
Bethesda, MD 20014
(301) 657-8220

Composition and Design
Muriel M. Furman
The Aerospace Corporation
Post Office Box 92957
Los Angeles, CA 90009
(213) 648-5405

PRELIMINARY

ADA

REFERENCE MANUAL

Additional copies of the 2 part set (Volume 14, Number 6, June 1979, Parts A & B) may be ordered prepaid from:

Association for Computing Machinery, Inc.
P.O. Box 12105
Church Street Station
New York, New York 10249

Price:	ACM and SIGPLAN Members	$ 16.00 prepaid
	All others	$ 22.00 prepaid

The Government of the United States of America grants free permission to reproduce this document for the purpose of evaluating and using the Ada language. The definition described by this document will be superseded by a final version in early 1980, at which time this preliminary version will be obsolete and should be retired.

THE SECRETARY OF DEFENSE
WASHINGTON, D. C. 20301

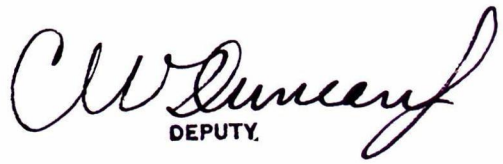
MAY 14 1979

The Department of Defense High Order Language Commonality program began in 1975 with the goal of establishing a single high order computer programming language appropriate for DoD embedded computer systems. This effort has been characterized by extensive cooperation with the European Community and NATO countries have been involved in every aspect of the work. The requirements have been distributed worldwide for comment through the military and civil communities, producing successively more refined versions. Formal evaluations were performed on dozens of existing languages concluding that a single language meeting these requirements was both feasible and desirable. Such a language has now been developed.

I wish to encourage your support and participation in this effort, and I submit the design of this language for your review and comment. Such comments and detailed justified change proposals should be forwarded to HOLWG, DARPA, 1400 Wilson Boulevard, Arlington, VA 22209 by 30 November 1979. The language, as amended by such response, will become a Defense standard in early 1980. Before that, changes will be made; after that, we expect that change will be minimal.

Beginning in May 1979, the effort will concentrate on the technical test and evaluation of the language, development of compilers and programming tools, and a capability for controlling the language and validating compilers. The requirements and expectations for the environment and the control of the language are being addressed in a series of documents already available to which comment is also invited. We intend that Government-funded compilers and tools will be widely and cheaply available to help promote use of the language.

Ada has been chosen as the name for the common language, honoring Ada Augusta, Lady Lovelace, the daughter of the poet, Lord Byron, and Babbage's programmer.



DEPUTY

Foreword

The language Ada is the result of a collective effort to design a language satisfying the Steelman requirements. The design team was led by Jean D. Ichbiah and has included Bernd Krieg-Brueckner, Brian A. Wichmann, Henry F. Ledgard, Jean-Claude Heliard, Jean-Raymond Abrial, John G.P. Barnes, and Olivier Roubine. In addition, major contributions were provided by G. Ferran, I.C. Pyle, S.A. Schuman, and S.C. Vestal.

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

At various stages of this project several persons had a constructive influence with their comments, criticisms and suggestions. They are P. Brinch Hansen, D.A. Fisher, G. Goos, C.A.R. Hoare, M. Woodger, and Mark Rain.

Over the two years spent on this project, three intense one-week design reviews were conducted with the participation of J.B. Goodenough, H. Harte, M. Kronental, K. Correll, R. Firth, A.N. Habermann, J. Teller, P. Wegner, and P. R. Wetherall.

These reviews, other comments by E. Boebert, P. Bonnard, T. Frogatt, H. Ganzinger, C. Hewitt, J.L. Mansion, F. Minel, E. Morel, J. Roehrich, A. Singer, D. Slosberg, and I.C. Wand; the numerous evaluation reports received on the preliminary design, the on-going work of the IFIP Working Group 2.4 on system implementation languages, and that of LTPL-E of Purdue Europe, all had a decisive influence on the final shape of the language Ada.

The work for this language definition was performed by Cii Honeywell Bull and Honeywell Systems & Research Center under contract with the United States Department of Defense. W.E. Carlson served as the Government's technical representative.

Table of Contents

1. Introduction

This report describes the Ada programming language, designed in accordance with the Steelman requirements of the United States Department of Defense. Overall, the Steelman requirements call for a language with considerable expressive power covering a wide application domain. As a result the language includes facilities offered by classical languages such as Pascal as well as facilities often found only in specialized languages. Thus the language is a modern algorithmic language with the usual control structures, and the ability to define types and subprograms. It also serves the need for modularity, whereby data, types, and subprograms can be packaged. It treats modularity in the physical sense as well, with a facility to support separate compilation.

In addition to these classical aspects, the language covers real time programming, with facilities to model parallel tasks and to handle exceptions. It also covers systems program applications. This requires access to system dependent parameters and precise control over the representation of data. Finally, both application level and machine level input-output are defined.

1.1 Design Goals

The Ada language was designed with three overriding concerns: a recognition of the importance of program reliability and maintenance, a concern for programming as a human activity, 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 Ada language requires that program variables be explicitly declared and that their type be specified. Automatic type conversion is prohibited. As a result, compilers can ensure that the types of objects satisfy their intended use. Furthermore, error prone notations have been avoided, and the language syntax 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 as within a unit.

Concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep the language as small as possible, given the ambitious nature of the application domain. We have attempted to cover this domain with a minimum number of underlying concepts integrated in a consistent and systematic way. Nevertheless we have tried to avoid the pitfalls of excessive involution, and in the constant search for simpler designs we have tried to provide language constructs with an intuitive mapping on what the user will normally expect.

No language can avoid the problem of efficiency. Languages that require overly 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 in the Ada language was examined in the light of present implementation techniques. Any proposed construct whose implementation was unclear or required excessive machine resources was rejected.

Perhaps most importantly, none of the above goals was considered something that could be achieved after the fact. The design goals drove the entire design process from the beginning.

1.2 Language Summary

A program in the Ada language is a sequence of higher level program units, which can be compiled separately. Program units may be subprograms (which define executable algorithms), package modules (which define collections of entities) or task modules (which define concurrent computations). The facility for separate compilation allows a program to be designed, written, and tested in largely independent parts. This facility is especially useful for large programs and the creation of libraries.

Program Units

A subprogram is the basic unit for expressing an algorithm. A subprogram can have parameters, which specify its connections to other program units. The Ada language distinguishes two kinds of subprograms: procedures and functions.

A procedure subprogram is the logical counterpart to a series of actions. For example, it may read in data, update variables, or produce some output. A function subprogram is the logical counterpart to a mathematical function for computing a value; unlike a procedure, a function can have no side effects. Value returning procedures are also allowed.

A package module is the basic unit for defining a collection of logically related entities. For example, packages can be used to define a common pool of data and types, a collection of related subprograms, or encapsulated types with associated operations. Portions of a package can be hidden from the user, thus allowing access only to the logical properties expressed by the package module.

A task module is similar to a package module, but with additional capabilities for parallel processing. Tasks may be implemented on multiple processors or with interleaved execution on a single processor. Synchronization is achieved by entries which are called like procedures but are executed in mutual exclusion. Like procedures, entries can contain parameters specifying the transmission of data between tasks.

Declarations and Statements

Each 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 define the execution of the program unit.

The declarative part associates names with declared entities. A name may denote a type, a constant, or a variable. A declarative part also introduces the names and parameters of other subprograms, task modules, and package modules to be used in the program unit.

Statements describe actions to be performed. An assignment statement specifies that the current value of a variable is to be replaced by a new value. A subprogram call statement invokes execution of a subprogram, after associating any arguments provided at the call with the corresponding ormal parameters of the subprogram.

If and case statements allow the selection of an enclosed sequence of statements based on the value of a condition or expression at the head of the statement.

The basic iterative mechanism in the language is the loop statement. A loop statement specifies that a sequence of statements is to be executed repeatedly until an iteration specification is completed or an exit statement is encountered.

Certain statements are only applicable to tasks. An initiate statement specifies that one or more tasks may begin execution concurrently with the initiating task. An entry call, which appears as a normal subprogram call, specifies that a task is ready for a rendezvous with another task containing the declaration of the entry. An accept statement within the other task specifies the actions (if any) to be executed when the corresponding entry is called. After the rendezvous is completed, both the calling task and the task containing the entry may continue their execution in parallel. A select statement allows a selective wait for one of several alternative rendezvous.

Execution of a program unit may lead to exceptional 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 the value of an uninitialized variable. To deal with these situations, the statements of a program unit can be textually followed by exception handlers describing the actions to be taken when the exceptional situation arises.

Data Types

Every object in the language has a type, which defines its logical properties and the operations that can be performed on objects of the type. There are four basic classes of types: scalar types, composite types, access types, and private types.

The scalar types INTEGER, BOOLEAN, and CHARACTER are predefined. Integer types provide a means of performing exact numerical computation. Approximate computation can be performed using floating point types (with a relative bound on the error) or using fixed point types (with an absolute bound on the error). Enumeration types provide a means for users to define problem dependent types with discrete values. Character sets can be defined as enumeration types.

Composite types allow definitions of structured objects with related components. The composite types in the language provide for 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. Alternative record structures can be defined by having a variant part within a record type.

Access types allow the construction of complex data structures that are created dynamically. Both the elements in the structure and their relation to other elements can be altered during program execution.

Private types can be defined in a package module that hides irrelevant structural details. Only the logically necessary properties are made visible to a user.

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

Other Facilities

Representation specifications can be used to specify the mapping between data types and features of an underlying machine. For example, the user can specify that an array is to be represented in packed form, that objects of a given type must be represented with a specified number of bits, or that the components of a record are to be represented in a specified storage layout.

Finally the language includes facilities for separate compilation, generic (that is, parameterized) program units, and both user level and machine level input-output.

1.3 Sources

A continual difficulty in language design is that one must both identify the capabilities required by the application domain and design language features that provide these capabilities.

The difficulty existed in this design, although to a much lesser degree than usual because of the Steelman requirements. These requirements often simplified the design process by permitting us to concentrate on the design of a given system satisfying a well defined set of capabilities, rather than on the definition of the capabilities themselves.

Another significant simplification of our design work resulted from earlier experience acquired by several successful Pascal derivatives developed with similar goals. These are the languages Euclid, Lis, Mesa, Modula, and Sue. Many of the key ideas and syntactic forms developed in these languages have a counterpart in the Ada language. We may say that whereas these previous designs could be considered as genuine research efforts, the Ada language is the result of a project in language design engineering, in an attempt to develop a product that represents the current state of the art.

Several existing languages such as Algol 68 and Simula and also recent research languages such as Alphard and Clu, influenced this language in several respects, although to a lesser degree than the Pascal family.

Finally, the evaluation reports on the initial formulation of the Green language, the other language proposals, and the general reviews had a significant effect on the language.

1.4 Syntax Notation

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

(a) Lower case words, some containing embedded underscores, denote syntactic categories, for example

    adding_operator

(b) Boldface words denote reserved words, for example

    array

(c) Square brackets enclose optional items, for example

    end loop [identifier];

(d) Braces enclose a repeated item. The item may appear zero or more times. Thus an identifier list is defined by

```
identifier_list ::= identifier {, identifier}
```

(e) Any syntactic category prefixed by an italicized word and an underscore is equivalent to the unprefixed corresponding category name. The prefix is intended to convey some semantic information. For example

*type*_name *task*_name

are both equivalent to the category

name

In addition, the syntax rules describing structured constructs are presented in a form that corresponds to the preferred paragraphing. For example, an if statement is defined as

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

2. Lexical Elements

This chapter defines the lexical elements of the language.

2.1 Character Set

All language constructs may be represented with a basic character set, which is subdivided as follows:

(a) Upper case letters
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(b) Digits
0 1 2 3 4 5 6 7 8 9

(c) Special characters
" # & ' ( ) * + , - . / : ; < = > _ |

(d) The space character

An extended character set, for example one including the following additional ASCII characters, may be used in programs:

(e) Lower case letters
a b c d e f g h i j k l m n o p q r s t u v w x y z

(f) Other special characters
! $ % ? @ [\ ] ^ ` { } ~

Every program may be converted into an equivalent program using only the basic character set. A lower case letter is equivalent to the corresponding upper case letter, except within character strings; rules for conversion of strings into the basic character set appear in section 2.5.

In addition, the following replacements are always allowed for characters that may not be available:

- the vertical bar character | is equivalent to the exclamation mark ! as a delimiter between choices (e.g. see 3.6.2). Note that on some terminals, the vertical bar appears as a broken vertical bar.
- the double quote character " is equivalent to a % character as a string bracket
- the sharp character # is equivalent to the colon : in a based number

2.2 Lexical Units and Spacing Conventions

The lexical units of a program are identifiers (including reserved words), numbers, strings, and delimiters. A delimiter is either one of the following special characters in the basic character set

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

or one of the following compound symbols

```
=>  ..  **  :=  =:  :=:  /=  >=  <=  <<  >>
```

Spaces may be inserted freely with no effect on meaning between lexical units. At least one space must separate adjacent identifiers or numbers. Besides terminating a comment, the end of each line is equivalent to a space. Thus each lexical unit must fit on one line.

2.3 Identifiers

Identifiers are used as names. Isolated underscore characters may be included and all characters are significant, including underscores.

```
identifier ::=
   letter {[underscore] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter
```

Note that identifiers differing only in the use of corresponding upper and lower case letters are considered as the same.

Examples:

COUNT	X	get_symbol	Ethelyn
SNOBOL_4	X1	page_count	STORE_NEXT_ITEM

2.4 Numbers

There are two classes of numbers: integers for exact computation, and real numbers for approximate computation. Their explicit representation is given here.

```
number ::= integer_number | approximate_number

integer_number ::= integer | based_integer

integer ::= digit {[underscore] digit}
```

```
based_integer ::=
   base # extended_digit {[underscore] extended_digit}

base ::= integer

extended_digit ::= digit | letter

approximate_number ::=
      integer.integer [E exponent]
   |  integer E exponent

exponent ::= [+] integer | - integer
```

Isolated underscore characters may be inserted between adjacent digits or extended digits of a number, but are not significant. Spaces may not appear within numbers.

Based integers can be represented with any base from 2 to 16. For bases above ten, digits may include the letters A through F with the conventional meaning 10 through 15.

Examples:

```
12    0    123_456                  -- integers

2#1011_0101    16#FFFF              -- based integers, values 181 and 65535

12.0    0.0    0.456    10_000.1    -- approximate numbers

1E6    1.0E-6    3.14e+3            -- approximate numbers with exponent
```

2.5 Character Strings

A character string is a sequence of zero or more characters prefixed and terminated by the string bracket character (the double quote " or its replacement the % character).

```
character_string ::= " {character} "
```

In order that arbitrary strings of characters may be represented, any included string bracket character must be written twice. The length of a string is the length of the sequence represented. Strings of length one are also used for literals of character types (see 3.5.1). Strings longer than one line must be represented using catenation.

Examples:

```
""                          -- an empty string
"*"    "A"    """"          -- three one-character literals * A "

"characters such as $, { and } may appear in strings"

"FIRST PART OF A STRING THAT " &
"CONTINUES ON THE NEXT LINE"
```

A character string may contain characters not in the basic character set. A string containing such characters can be converted to a string written with the basic character set by using identifiers denoting these characters in catenated strings. Such identifiers are defined in the predefined environment. Thus the string "AB$CD" could be written as "AB" & DOLLAR & "CD". Similarly, the string "ABcd" with lower case letters could be written as "AB" & LC_C & LC_D.

2.6 Comments

A comment starts with a double hyphen and is terminated by the end of the line. It may only appear following a lexical unit or at the beginning or end of a program unit. Comments have no effect on the meaning of a program; their sole purpose is the enlightenment of the human reader.

Examples:

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

end; -- processing of LINE is complete

-- a stand alone comment
-- and its continuation
```

2.7 Pragmas

Pragmas are used to convey information to the compiler. A pragma begins with the reserved word **pragma** followed by the name of the pragma. A pragma can have arguments, which can be identifiers, strings, or numbers.

```
pragma ::=
   pragma identifier [(argument {, argument})];

argument ::= identifier | character_string | number
```

Pragmas may appear before a program unit, and wherever a declaration or a statement may appear. The extent of the effect of a pragma depends on the pragma.

A pragma may be language defined or implementation defined. All language defined pragmas are described in Appendix B.

Examples of pragmas:

```
pragma LIST(OFF);                    -- suppress listing
pragma OPTIMIZE(TIME);               -- optimization specification
pragma INCLUDE("COMMON_TEXT");       -- include text file
pragma DEBUG(ON);                    -- set debugging mode on
```

2.8 Reserved Words

The identifiers listed below are called *reserved words* and are reserved for special significance in the language. As such, these identifiers may not be declared by the programmer. For readability of this manual, the reserved words appear in lower case boldface.

abort	**declare**	**generic**	**of**	**select**
accept	**delay**	**goto**	**or**	**separate**
access	**delta**		**others**	**subtype**
all	**digits**		**out**	
and	**do**	**if**		
array		**in**	**package**	**task**
assert		**initiate**	**packing**	**then**
at		**is**	**pragma**	**type**
	else		**private**	
	elsif		**procedure**	
	end	**loop**		**use**
begin	**entry**		**raise**	
body	**exception**	**mod**	**range**	
	exit		**record**	**when**
			renames	**while**
		new	**restricted**	
case	**for**	**not**	**return**	
constant	**function**	**null**	**reverse**	**xor**

3. Declarations and Types

This chapter describes the types in the language and the rules for declaring constants and variables.

3.1 Declarations

A declaration associates an identifier with a declared entity. Each identifier must be declared before it is used, with the exception of labels. There are several kinds of declarations.

```
declaration ::=
      object_declaration          | type_declaration
    | subtype_declaration         | private_type_declaration
    | subprogram_declaration      | module_declaration
    | entry_declaration           | exception_declaration
    | renaming_declaration
```

The process by which a declaration achieves its effect is called the *elaboration* of the declaration. Any expression appearing in a declaration is evaluated when the declaration is elaborated unless otherwise stated.

Object, type, and subtype declarations are described here. The remaining declarations are described in later chapters.

3.2 Object Declarations

An object is a variable or a constant. An object declaration introduces one or more named objects of a given type. These objects can only have values of this type.

```
object_declaration ::=
   identifier_list : [constant] type [:= expression];

identifier_list ::= identifier {, identifier}
```

An object declaration may include an expression which specifies the initial value of the declared objects. This expression is evaluated and its value is assigned to each of the declared objects, as part of the elaboration of the object declaration.

An object is a constant if its declaration includes the reserved word **constant**. The value of a constant cannot be modified. If a constant object has components, they cannot be modified.

It is possible to defer the initialization of a constant record component (see 3.7.1) and of a constant of a private type declared in the visible part of a module (see 7.4).

Examples of variable declarations:

```
ITEM_1, ITEM_2    : INTEGER;
SORT_COMPLETED    : BOOLEAN := FALSE;
OPTION_TABLE      : array (1 .. N) of OPTION;
```

Examples of constant declarations:

```
LIMIT         : constant INTEGER  := 10_000;
LOW_LIMIT     : constant INTEGER  := LIMIT/10;
TOLERANCE     : constant FLOAT    := SQRT(X);
LENGTH        : constant INTEGER; -- deferred initialization;
```

Notes:

The expression initializing a constant may (but need not) be a static expression (see 4.8). In the above examples, LIMIT and LOW_LIMIT are initialized with static expressions, but TOLERANCE is not.

3.3 Type and Subtype Declarations

A type characterizes a set of values and a set of operations applicable to those values. The values are denoted by literals or aggregates of the type, or can be obtained as the results of operations. The operations and the properties of the values are said to be *attributes* of the type. Any subprogram with a parameter or result of the type is an attribute of the type.

There exist several classes of types. *Scalar* types are types whose values have no components; they comprise types defined by enumeration of their values, integer types, and real types. *Array* and *record* types are composite; their values consist of several component values. An *access* type is a type whose values provide access to other objects. The attributes resulting from the definition of these classes of types are predefined attributes (see 4.1.3). Finally, there are *private* types where the set of possible values is clearly defined, but not known to the users of such types. Hence, a private type is only known by the set of operations applicable to its values (see 7.4).

The set of possible values of a type can be restricted without changing the set of applicable operations. Such a restriction is called a *constraint*. A value is said to belong to a *subtype* of a given type if it obeys such a constraint. Naturally, subtypes may not be found for user defined private types since nothing is known *a priori* about the set of possible values.

```
type ::= type_definition | type_mark [constraint]

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

```
type_mark ::= type_name | subtype_name

constraint ::=
      range_constraint   | accuracy_constraint
    | index_constraint   | discriminant_constraint

type_declaration ::=
   type identifier [is type_definition];

subtype_declaration ::=
   subtype identifier is type_mark [constraint];
```

Every type definition introduces a distinct type. A type declaration associates a name with a type. A subtype declaration introduces a name as an abbreviation for a type name with some possible constraint. Each constraint is evaluated when the declaration in which it appears is elaborated.

An incomplete type declaration of the form

```
type T;
```

is used for the declaration of mutually dependent access types (see 3.8); the complete type declaration must follow in the same declarative part.

Examples of type declarations:

```
type COLOR     is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
type COL_NUM   is range 1 .. 72;
type TABLE     is array (1 .. 10) of INTEGER;
```

Examples of subtype declarations:

```
subtype RAINBOW    is COLOR range RED .. BLUE;
subtype SMALL_INT  is INTEGER range -10 .. 10;
subtype ZONE       is COL_NUM range 1 .. 6;
```

Notes:

Two type definitions always introduce two distinct types, even if they are textually identical. For example, the enumeration type definitions given in the declarations of A and B below define distinct types, although the set of values of one of them is a copy of the set of values of the other.

```
A  : (ON, OFF);
B  : (ON, OFF);
```

On the other hand, C and D in the following declaration are of the same type, since only one type definition is given.

```
C, D : (WHITE, GREY, BLACK);
```

3.4 Derived Type Definitions

A derived type definition introduces a new type deriving its characteristics from those of an existing type.

derived_type_definition ::= **new** type_mark [constraint]

With a type declaration of the form:

type NEW_TYPE **is new** OLD_TYPE;

the new type derives all its characteristics from those of the old type:

- The new type belongs to the same class of types as the old type (for example, the new type is a record type if the old type is) and the same attributes are predefined.
- The set of values of the new type is a copy of the set of values of the old type. The constraints associated with the old type apply to objects of the new type.
- The notation for literals or aggregates of the new type is the same as for the old. Such literals and aggregates are said to be *overloaded*. The notation used to denote components of objects of the new type is the same as for the old.
- For each visible subprogram attribute of the old type, a subprogram attribute of the new type is derived in which occurrences of the name of the old type are in effect replaced by the name of the new type. Such subprograms are said to be *overloaded*. Assignment is available for the new type if it is for the old.
- Any explicit representation specification (see 13) given for the old type also applies to the new type.

The effect of such a type declaration is thus to create a new type distinct from the old type, but equivalent in effect to what would be obtained by duplicating the old type definition and all its applicable operations. Explicit conversions are allowed between the old type and the new type (see 4.6.2).

A type declaration of the form:

type NEW_TYPE **is new** OLD_TYPE *constraint*;

is equivalent to the succession of declarations:

type *new_type* **is new** OLD_TYPE;
subtype NEW_TYPE **is** *new_type constraint*;

where *new_type* is an identifier distinct from those of the program. Hence, the values and operations of the old type are inherited by the new type, but objects of the new type must satisfy the added constraint.

3.5 Scalar Types

Scalar types comprise discrete types and real types. Discrete types are the enumeration types and integer types; they may be used for indexing and iteration over loops. Numeric types are the integer and real types. All scalar types are ordered. A range constraint specifies a subset of values of the type or subtype.

range_constraint ::= **range** range

range ::= simple_expression .. simple_expression

The range L .. R describes the values from L to R inclusive. An empty range is a range for which L is greater than R. The type of the simple expressions in a range constraint is the type for which the range constraint is specified.

Predefined Attributes

For any scalar type or subtype T, the following attributes are predefined (see also 4.1.3 and Appendix A):

T'FIRST the minimum value of the type or subtype T

T'LAST the maximum value of the type or subtype T

For every discrete type or subtype T, the subprogram attributes T'SUCC, T'PRED, and T'ORD are predefined as follows:

T'SUCC(X) the value succeeding the value X in T

T'PRED(X) the value preceding the value X in T

T'ORD(X) the ordinal position of the value X in T. For example T'ORD(T'FIRST) = 1

The exception RANGE_ERROR is raised by the function call T'SUCC(T'LAST) and similarly by T'PRED(T'FIRST).

3.5.1 Enumeration Types

An enumeration type definition introduces a set of values by listing the values.

enumeration_type_definition ::=
  (enumeration_literal {, enumeration_literal})

enumeration_literal ::= identifier | character_literal

An enumerated value is represented by an identifier or a character literal. Hence, a character set can be defined by an enumeration type. Order relations between enumeration vaues follow the order of listing, the first being less than the last.

Within a sequence of declarations, an enumeration literal can appear in different enumeration types. Such enumeration literals are said to be *overloaded*. When ambiguities arise in the use of such literals they can be resolved by providing an explicit qualification (see 4.6).

Examples:

```
type DAY     is (MON, TUE, WED, THU, FRI, SAT, SUN);
type SUIT    is (CLUBS, DIAMONDS, HEARTS, SPADES);
type HEXA    is ('A', 'B', 'C', 'D', 'E', 'F');
type COLOR   is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN, BLACK);
type LIGHT   is (RED, AMBER, GREEN);  -- RED and GREEN are overloaded

subtype WEEK_DAY  is DAY    range MON .. FRI;
subtype MAJOR     is SUIT   range HEARTS .. SPADES;
subtype RAINBOW   is COLOR  range RED .. BLUE;     -- the color RED, not the light
```

3.5.2 Character Types

A character type is an enumeration type that contains character literals and possibly identifiers. The predefined type CHARACTER denotes the full ASCII character set of 128 characters (see Appendix C).

3.5.3 Boolean Type

There is a predefined enumeration type named BOOLEAN. It contains the two literals FALSE and TRUE ordered with the relation FALSE < TRUE. The evaluation of conditions delivers results of this predefined type.

3.5.4 Integer Types

The predefined type named INTEGER denotes a subset of the integers. Other integer types can be introduced by integer type definitions or can be derived from the type INTEGER.

integer_type_definition ::= range_constraint

The range of integer numbers is implicitly limited by the representation adopted by an individual implementation. An implementation may have predefined types such as SHORT_INTEGER and LONG_INTEGER, which have respectively shorter and longer ranges than INTEGER.

A type declaration of the form

type T **is range** L .. R;

where L and R denote integer values, introduces an integer type equivalent to

type T **is new** *integer_type* **range** L .. R;

where the *integer_type* is implicitly chosen so as to contain the values L through R and is one of the predefined types such as SHORT_INTEGER, INTEGER, or LONG_INTEGER.

Examples:

```
type PAGE_NUM   is range 1 .. 2_000;
type LINE_SIZE  is new INTEGER range 1 .. MAX_LINE_SIZE;

subtype SMALL_INT    is INTEGER   range -10 .. 10;
subtype COLUMN_PTR   is LINE_SIZE range 1 .. 10;
```

Notes:

The smallest integer value supported by the implementation is SYSTEM'MIN_INT and the largest value SYSTEM'MAX_INT.

3.5.5 Real Types

Real types provide approximations to the real numbers, with relative bounds on errors for floating point types, and with absolute bounds on errors for fixed point types.

```
real_type_definition ::= accuracy_constraint

accuracy_constraint ::=
      digits simple_expression [range_constraint]
    | delta simple_expression [range_constraint]
```

For floating point types the error bound is specified as a relative precision by giving the minimum number of decimal digits for the mantissa.

A given implementation can have predefined floating point types, such as SHORT_FLOAT, FLOAT, and LONG_FLOAT, which correspond to the hardware supplied floating point types. Real type definitions of the forms

```
digits P
digits P range L .. R
```

where P is a static integer expression (see 4.8) specifying a number of decimal digits, and where L and R are floating point values, are equivalent to the type definitions

```
new floating_point_type digits P
new floating_point_type digits P range L .. R
```

where *floating_point_type* is implicitly chosen as an appropriate predefined floating point type. The implemented precision must be at least that of the precision specified in the corresponding definition. If a range is provided, it must be covered by the chosen predefined type.

For fixed point types, the error bound is specified as an absolute value, called the *delta* of the fixed point type. The implemented error bound must be at least as fine as the specified delta. In a fixed point type definition, the range constraint cannot be omitted, since this determines the representation to be used for values of the type; the expressions specifying the range and the delta must be static expressions.

In a subtype or object declaration, an accuracy constraint can be applied to a previously declared real type. For a fixed point type, the delta of the constraint cannot be smaller than the delta of the type. For a floating point type, the number of digits specified in the constraint cannot be larger than that of the type. In all cases, the delta or the digits must be given by static expressions.

Examples:

```
type COEFFICIENT is digits 10 range -1.0 .. 1.0;

type VOLT  is delta 0.125 range 0.0 .. 255.0;
type FRAC  is delta 0.00001 range 0.0 .. 1.0;

type MASS is new FLOAT digits 7 range 0.0 .. 1E30;

subtype S_VOLT is VOLT delta 0.5;  -- same range as VOLT
subtype COEFF  is COEFFICIENT digits 8;
```

Predefined Attributes

For a floating point type or subtype T, the following attributes are predefined:

T'DIGITS	the specified number of digits (it is of type INTEGER)
T'SMALL	the smallest positive value expressible with the representation and precision of type T
T'LARGE	the largest positive value expressible with the representation and precision of type T

For a fixed point type or subtype T, the following attribute is predefined:

T'DELTA	the value of the specified delta

For any real type T the following attribute is predefined:

T'BITS	the minimum number of bits needed for the representation of the mantissa of T

3.6 Array Types

An array object is a set of components of the same component type. A component of an array is designated using one or more index values belonging to specified discrete types.

```
array_type_definition ::=
   array (index {, index}) of type_mark [constraint]

index ::= discrete_range | type_mark

discrete_range ::= [type_mark range] range

index_constraint ::= (discrete_range {, discrete_range})
```

An array object is characterized by the number of indices, the type of each index, the lower and upper bound for each index, and the type and possible constraints of the components. In an array type definition, each index can be specified either by a discrete range or by a type mark. These two forms of index specifications have different consequences:

(1) *Index specified by a discrete range*

For all objects of the array type, the discrete range determines both the permitted type for the index values and the lower and upper bound for the index values.

(2) *Index specified by a type mark*

For all objects of the array type, the type mark only determines the permitted type for the index values. The actual values of the lower and upper bound of the index considered can be different for different objects of the array type.

The bounds must be given for each array separately in its object declaration by an index constraint, or can be obtained from the initial value. For an array formal parameter, the bounds are obtained from the actual parameter.

For a multi-dimensional array, if one index position is specified by a discrete range, all index positions must be specified by discrete ranges. Similarly, an index constraint must provide ranges for all index positions. For accessing components, an *n*-dimensional array is equivalent to a one-dimensional array of (*n*-1)-dimensional subarrays.

If the bounds of a discrete range are integer numbers, these are assumed to be of the predefined type INTEGER if their type is not otherwise known from the context.

Predefined Attributes

For an array object A (or for an array type A with specified bounds), the following attributes are predefined (*i* is an integer value):

A'FIRST	the lower bound of the first index
A'LAST	the upper bound of the first index
A'LENGTH	the number of components of the first index (zero when no components)
A'FIRST(*i*)	the lower bound of the *i*-th index
A'LAST(*i*)	the upper bound of the *i*-th index
A'LENGTH(*i*)	the number of components of the *i*-th index

Examples of array types with specified bounds:

```
type TABLE     is array (1 .. 10) of INTEGER;
type SCHEDULE  is array (DAY'FIRST .. DAY'LAST) of BOOLEAN;
type LINE      is array (1 .. MAX_LINE_SIZE) of CHARACTER;
```

Examples of array types with unspecified bounds:

```
type MATRIX      is array (INTEGER, INTEGER) of REAL;
type BIT_VECTOR  is array (INTEGER) of BOOLEAN;
```

Examples of array declarations:

```
GRID      : array (1 .. 80, 1 .. 100) of BOOLEAN;
MIX       : array (COLOR range RED .. GREEN) of BOOLEAN;
MY_TABLE  : TABLE;  -- all arrays of type TABLE have the same length
```

```
BOARD     : MATRIX (1 .. 8,  1 .. 8);
RECTANGLE : MATRIX (1 .. 20, 1 .. 30);

-- MY_TABLE'FIRST = 1,   BOARD'LAST(1) = 8,   RECTANGLE'LAST(2) = 30
```

3.6.1 Index Ranges of Arrays

The range of each index of an array must be known when the declaration of the array is elaborated (or when allocated in the case of access types). The expressions defining the range of an index need not be static, but can depend on computed results. Such arrays are called *dynamic arrays*. In records, dynamic arrays may only appear when the dynamic bounds are discriminants of the record type.

Examples:

```
SQUARE : MATRIX (1 .. N, 1 .. N);      -- N need not be static

type VAR_LINE is access
   record
      LENGTH  : constant INTEGER;      -- value given on initialization
      IMAGE   : STRING(1 .. LENGTH);
   end record;
```

3.6.2 Aggregates

An aggregate denotes an array or record value constructed from component values.

```
aggregate ::=
   (component_association {, component_association})

component_association ::=
   [choice {| choice} => ] expression

choice ::= simple_expression | discrete_range | others
```

The expressions define the values to be associated with components. They can be given by position (in index order for array components, in textual order for record components) or by naming the chosen components (with index values for array components, with the corresponding identifiers for record components). An aggregate defining the value of an object must provide values for all components of the object.

For named components, the expressions can be given in any order, but if both notations are used in one aggregate, the positional component associations must be given first.

A choice given as a discrete range stands for all index values in the range. The choice **others** stands for all components not specified by previous choices and can only appear last. Choices with discrete values are also used in variant parts of records and in case statements. Each choice may only appear once in an aggregate (variant part or case statement) and, except for the choice **others**, its value must be determinable statically.

When an aggregate used as an initial value is expected to provide the bounds of an array object, the choice **others** cannot be used. For an array whose index is only specified by a type mark T, the lower bound is assumed to be equal to T'FIRST if the initialization is given by a positional aggregate.

An aggregate for an *n*-dimensional array is written as a one-dimensional aggregate of components that are (*n*-1)-dimensional array values.

Examples:

```
A   : TABLE := (7,9,5,1,3,2,4,8,6,0);                -- A (1) = 7, A(10) = 0
B   : TABLE := (5,4,8,1, others => 20);              -- B (1) = 5, B(10) = 20
C   : TABLE := (2 | 4 | 10 => 1, others => 0); -- C (1) = 0, C(10) = 1

NULL_MATRIX : constant MATRIX := (1 .. 10 => (1 .. 10 => 0.0));
-- NULL_MATRIX'FIRST(1) = 1,   NULL_MATRIX'LAST(2) = 10

ENGLISH_SCHOOL_DAYS  : constant SCHEDULE := (MON .. FRI => TRUE,   others => FALSE);
FRENCH_SCHOOL_DAYS   : constant SCHEDULE := (WED | SUN => FALSE, others => TRUE);
```

3.6.3 Strings

The predefined type STRING denotes one-dimensional arrays of the predefined type CHARACTER, indexed by values of the predefined subtype NATURAL:

```
subtype NATURAL is INTEGER range 1 .. INTEGER'LAST;
type STRING is array (NATURAL) of CHARACTER;
```

Character strings (see 2.5) are a special form of aggregate applicable to the type STRING and other one-dimensional arrays of characters.

Catenation is a predefined operator over one-dimensional arrays, and is represented as &. For strings, it corresponds to the following function:

```
function "&" (X, Y : STRING) return STRING is
   S : STRING(1 .. X'LENGTH + Y'LENGTH);
begin
   S(1 .. X'LENGTH) := X;
   S(X'LENGTH + 1 .. S'LAST) := Y;
   return S;
end;
```

Examples:

```
BLANKS   : STRING(1 .. 120)  := (1 .. 120 => ' ');
MESSAGE  : constant STRING   := "HOW MANY CHARACTERS?";
-- MESSAGE'FIRST = 1,   MESSAGE'LAST = number of characters

SAY_TWICE : constant STRING := MESSAGE & MESSAGE & CR & LF;
```

3.7 Record Types

A record object is a structure with named components. A record type definition can include a variant part denoting alternative record structures.

```
record_type_definition ::=
   record
      component_list
   end record

component_list ::=
   {object_declaration} [variant_part] | null;

variant_part ::=
   case discriminant of
      {when choice {| choice} =>
         component_list}
   end case;

discriminant ::= constant_component_name
```

The components of a record are defined by object declarations. Components can be of different types. The value of an expression provided as a component initialization is evaluated when the record type definition is elaborated. This value is used to initialize the corresponding component for every record declared of this type.

Recursion in record type definitions is not allowed unless an intermediate access type is used (see 3.8).

Examples:

```
type DATE is
   record
      DAY    : INTEGER range 1 .. 31;
      MONTH  : MONTH_NAME;
      YEAR   : INTEGER range 0 .. 2000;
   end record;

type COMPLEX is
   record
      RE  : FLOAT := 0.0;
      IM  : FLOAT := 0.0;
   end record;

-- both components of every complex record are initialized to zero
```

3.7.1 Constant Record Components and Discriminants

A constant component of a record which is not given an explicit value in the type definition is a deferred constant. Such a deferred constant can only be assigned by means of a complete record assignment.

A record component can be a dynamic array only if the bounds that are not static are deferred constant components of the record type. A deferred constant component used in this way or in a variant part is called a *discriminant* of the record type.

The only permissible dependencies between record components are the dependencies of an array bound and of a variant on a discriminant.

Example:

```
type BUFFER is
  record
    LENGTH   : constant INTEGER range 1 .. MAX_LENGTH; -- the discriminant
    POS      : INTEGER range 0 .. MAX_LENGTH := 0;
    IMAGE    : array (1 .. LENGTH) of CHARACTER;
  end record;
```

3.7.2 Variant Parts

A record type with a variant part specifies alternative record components. Each variant defines the components for the corresponding value of the discriminant. A variant can have an empty component list, which must be specified by **null**.

Example:

```
type PERIPHERAL is
  record
    STATUS   : (OPEN, CLOSED);
    UNIT     : constant (PRINTER, DISK, DRUM);    -- the discriminant
    case UNIT of
      when PRINTER =>
        LINE_COUNT : INTEGER range 1 .. PAGE_SIZE;
      when others =>
        CYLINDER : CYLINDER_INDEX;
        TRACK    : TRACK_NUMBER;
    end case;
  end record;
```

3.7.3 Record Aggregates and Discriminant Constraints

An aggregate is used to provide values for all the components of a record. In an aggregate for a record variant, the discriminant value must be a static expression and must appear before the values for the corresponding components of the variant part.

discriminant_constraint ::= aggregate

A discriminant constraint is used to constrain discriminants of a record to specific values; it is expressed as an aggregate specifying values for discriminants only. Discriminant constraints may be used to define subtypes of record types with variants.

Examples of record aggregates:

```
(4, JULY, 1776)
(DAY => 4, MONTH => JULY, YEAR => 1776)
(STATUS => CLOSED, UNIT => DISK, CYLINDER => 9, TRACK => 1)
```

Examples of record variable declarations with constraints:

```
WRITER  : PERIPHERAL(PRINTER);

CARD    : BUFFER(LENGTH => 80);
```

Example of record subtype:

```
subtype DISK_DEVICE is PERIPHERAL(UNIT => DISK);
```

3.8 Access Types

Objects declared in a program are accessible by their name. They exist during the lifetime of the declarative part to which they are local. In contrast, objects may also be created dynamically by the execution of *allocators* (see 4.7). Since they do not occur in an explicit object declaration, they cannot be designated by their name. Instead, access to such an object is achieved by an *access value* returned by an allocator.

access_type_definition ::= **access** type

An access type definition characterizes a set of access values which may be used to designate objects of the type mentioned after the reserved word **access**. This type cannot be another access type. The dynamically created objects designated by the values of an access type form a collection implicitly associated with the type. An access value obtained from an allocator can be assigned to several access variables. Hence a given dynamically created object may be designated by more than one variable or constant of the access type. The access value **null** belongs to every access type and designates no object at all. It may be used to initialize access variables.

An object of an access type that is introduced as a constant cannot have its value changed, nor can the value of the designated object be changed. Such an access object can only be used in an expression or as an **in** parameter; it cannot be assigned to an access variable (otherwise the designated object could be modified using the variable).

Constraints specified for an access type apply to the type of the designated objects. Qualification of an expression of an access type applies to the designated object.

Although the dynamically created objects may not be of an access type, there is no restriction on their components. Thus, components of the object designated by the values of an access type may be values of the same or of another access type. This permits recursive and mutually dependent access types (whose declaration requires a prior incomplete type declaration for one or more types).

Examples:

```
type TEXT is access BUFFER;

type LIST_ITEM is access
   record
      VALUE   : INTEGER;
      SUCC    : LIST_ITEM;
      PRED    : LIST_ITEM;
   end record;

HEAD : LIST_ITEM := null;
```

Example of mutually dependent access types:

```
type CAR;                    -- a prior incomplete type declaration

type PERSON is access
   record
      NAME    : STRING(1 .. 20);
      AGE     : INTEGER range 0 .. 130;
      SPOUSE  : PERSON;
      VEHICLE : CAR;    -- valid because CAR declared above
   end record;

type CAR is access           -- the complete type declaration
   record
      NUMBER  : INTEGER;
      OWNER   : PERSON;
   end record;

MY_CAR, YOUR_CAR, NEXT_CAR : CAR;
```

4. Names, Variables, and Expressions

4.1 Names

Names denote declared entities such as variables, constants, types, and program units.

```
name ::=
      identifier              | indexed_component
    | selected_component      | predefined_attribute

indexed_component  ::= name(expression {, expression})

selected_component ::= name . identifier

predefined_attribute ::= name ' identifier
```

The simplest form for the name of an entity is the identifier given in its declaration.

Examples of simple names:

```
INDEX       -- the name of a scalar variable
GRID        -- the name of an array variable
FLOAT       -- the name of a type
SQRT        -- the name of a function
OVERFLOW    -- the name of an exception
```

4.1.1 Indexed Components

An indexed component can denote either

(a) A component of an array:

The name identifies the array, (or an access object whose value designates the array, see 3.8) and the expressions give the indices for the component. If the array has more dimensions than the given number of expressions, the array component is a subarray of the named array.

(b) A task in a family of tasks:

The name identifies the task family and the expression (only one can be given) specifies the index of the individual task.

(c) An entry in a family of entries:

The name identifies the entry family and the expression (only one can be given) specifies the index of the individual entry.

If evaluation of one of the expressions gives an index value that is outside the range specified for the index, the exception INDEX_ERROR is raised.

Examples of indexed components:

```
GRID(1, J+1)          -- an array component
GRID(2)               -- a subarray of GRID
PRINTER(I)            -- a task in the task family PRINTER
```

4.1.2 Selected Components

A selected component can denote either

(a) A component of a record:

The name identifies the record (or an access object whose value designates the record) and the identifier specifies the record component.

(b) An entity declared in the visible part of a module:

The name identifies the module and the identifier specifies the declared entity.

(c) An entity declared in an enclosing unit:

The name identifies the enclosing unit and the identifier specifies the declared entity.

(d) A user-defined attribute of a type:

The name identifies the type and the identifier specifies a user-defined subprogram attribute of the type (see 3.3).

For variant records, a component identifier can denote a component in a variant part. In such a case, the selected component must belong to the variant prescribed by the discriminant of the record, otherwise the exception DISCRIMINANT_ERROR is raised.

Examples of selected components:

```
APPOINTMENT.DAY              -- a record component
NEXT.SUCC.VALUE              -- a record component
KEY_TABLE(SYMBOL).LENGTH     -- a record component

DEVICE.READ                  -- an entry of the task DEVICE
PRINTER(I).WRITE             -- an entry of the task PRINTER(I)
TABLE_MANAGER.INSERT         -- a procedure in the package TABLE_MANAGER

MAIN.ITEM_COUNT              -- a variable declared in the procedure MAIN
STACK.MAX_SIZE               -- an attribute of the type STACK
```

Notes:

For a record structure with nested record structures, the name of each level must be given to name a nested component.

4.1.3 Predefined Attributes

For user-defined attributes, as explained above, the notation of selected components is used; the named entity is a type, and the attribute identifier is a subprogram provided by the user. For predefined attributes, the apostrophe notation is used; the named entity need not be a type and the attribute identifier is predefined in the language. A predefined attribute identifier is always prefixed by an apostrophe, hence these identifiers are not reserved. Specific predefined attributes are described with the corresponding language constructs.

Appendix A gives a list of all the language predefined attributes. Additional predefined attributes may exist for an implementation.

Examples of predefined attributes:

```
COLOR'FIRST          -- minimum value of the enumeration type COLOR
FLOAT'DIGITS         -- precision of the real type FLOAT
GRID'LAST(2)         -- upper bound of the second dimension of GRID
PRINTER(I)'ACTIVE    -- TRUE if the task PRINTER(I) is active
DATE'SIZE            -- number of bits for records of type DATE
CARD'ADDRESS         -- address of the record variable CARD
```

4.2 Literals

A literal denotes an explicit value of a given type.

```
literal ::=
   number | enumeration_literal | character_string | null
```

A number or an enumeration literal denotes a value of the corresponding scalar type. The access value **null** designates no object at all. Literals for approximate numbers are rounded to the precision required by the context in which they are used.

Examples:

```
3.14159_26536        -- an approximate number
1_345                -- an integer number
CLUBS                -- an enumeration literal
"A"                  -- a character literal, also a character string
"SOME TEXT"          -- a character string
```

4.3 Variables

A variable is an object of an arbitrary type whose value can be changed. A variable can be a scalar, an array, a record, or an access object. Alternatively, it can be a component of another object or a slice of an array.

```
variable ::= name [(discrete_range)] | name.all
```

When a name is followed by a discrete range, the name must be the name of an array (or subarray) or of an access object whose value designates an array. The range must denote a contiguous sequence of index values for the first dimension of the array (or subarray). Such a variable is called an array slice. Its type is that of the array, with the constraint given by the discrete range. For names of access objects with the qualifier **all**, the variable denotes the entire object designated by the access value.

Examples:

```
X                    -- a scalar variable
GRID                 -- an array variable
GRID(I, J)           -- a component of the array variable GRID
GRID(1 .. 10)        -- a slice of the array GRID
GRID(2)(1 .. 10)     -- a slice of the subarray GRID(2)

TODAY.MONTH          -- a record component
NEXT_CAR.OWNER       -- a component of the record designated by NEXT_CAR
NEXT_CAR.all         -- the entire record value designated by NEXT_CAR
```

4.4 Expressions

An expression is a formula that defines the computation of a value.

```
expression ::=
      relation {and relation}
    | relation {or relation}
    | relation {xor relation}

relation ::=
      simple_expression [relational_operator simple_expression]
    | simple_expression [not] in range
    | simple_expression [not] in type_mark [constraint]

simple_expression ::= [unary_operator] term {adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary]

primary ::=
      literal | aggregate | variable | allocator
    | subprogram_call | qualified_expression | (expression)
```

Primaries include constants and predefined attributes, which are covered by the syntactic category "variable". An expression of a given type is also regarded as a one-component aggregate for a corresponding array or record type. The type of an expression depends on the type of its constituents, as described below.

Examples of primaries:

```
4.0                     -- number
(1 .. 10 => 0)          -- aggregate array value
VOLUME                  -- value of a variable
DATE'SIZE               -- predefined attribute
SINE(X)                 -- function subprogram call
REAL(I*J)               -- qualified expression
(LINE_COUNT + 10)       -- parenthesized expression
```

Examples of expressions:

```
VOLUME                              -- primary
B**2                                -- factor
LINE_COUNT mod PAGE_SIZE            -- term

-4.0                                -- simple expression
not DESTROYED                       -- simple expression
B**2 - 4.0*A*C                      -- simple expression

PASSWORD(1 .. 5) = "JAMES"          -- relation
I not in 1 .. 10                    -- relation

INDEX = 0 or ITEM_HIT               -- expression
(COLD and SUNNY) or WARM            -- expression
```

4.5 Operators and Expression Evaluation

The operators in the language are grouped into six classes, given in the following order of increasing precedence:

```
logical_operator          ::=  and | or | xor

relational_operator       ::=  =  | /= | < | <= | > | >=

adding_operator           ::=  +  | -  | &

unary_operator            ::=  +  | -  | not

multiplying_operator      ::=  *  | /  | mod

exponentiating_operator   ::=  **
```

For a sequence of operators of the same precedence level, evaluation proceeds in textual order from left to right, or in any order giving the same result. The primaries of an expression are also evaluated in textual order. All primaries are evaluated and all operations are performed.

The operands, result types, and the meaning of the predefined operators are given below. Note that some operations may result in exception conditions for some values of the operands (see chapter 11). Real expressions are not necessarily calculated with exactly the specified accuracy (precision or delta), but the accuracy used will be at least as good as that specified.

Examples of precedence:

```
not SUNNY or WARM        -- same as (not SUNNY) or WARM
X > 4.0 and Y > 0.0      -- same as (X > 4.0) and (Y > 0.0)

-4.0*A**2                -- same as -(4.0 * (A**2))
Y**(-3)                  -- parentheses are necessary
A / B * C                -- same as (A/B)*C
```

4.5.1 Logical Operators

Logical operators are applicable to boolean values and to one dimensional arrays of boolean values having the same number of components. The operations on arrays are performed on a component by component basis.

Operator	*Operation*	*Operand Type*	*Result Type*
and	conjunction	BOOLEAN boolean array type	BOOLEAN same array type
or	inclusive disjunction	BOOLEAN boolean array type	BOOLEAN same array type
xor	exclusive disjunction	BOOLEAN boolean array type	BOOLEAN same array type

4.5.2 Relational and Membership Operators

The relational operators have operands of the same type and return boolean values. Note that equality and inequality are defined for any two objects of the same type (unless the type is a restricted type, see 7.4).

Operator	*Operation*	*Operand Type*	*Result Type*
= /=	equality and inequality	any type	BOOLEAN
< <= > >=	test for ordering	any scalar type	BOOLEAN

Equality for the discrete types is equality of the values. For a floating (or fixed) point type T, if two values differ by less than T'SMALL (or T'DELTA), then the result delivered by a relational operator is implementation defined. Equality for array and record types is equality of the components, as given by the predefined operators. Hence, this operation is unchanged by any redefinition of equality on the component types involved. Two access values (see 3.8) are equal if they designate the same dynamically allocated object.

The inequality operator gives the complementary result to the equality operator.

The membership operators **in** and **not in** test for membership of a value of any type within a corresponding range, subtype, or constraint. These operators return a boolean value and have the same precedence as the relational operators.

Examples:

```
X /= Y        -- with real X and Y, is implementation defined

MY_CAR = null                -- true if MY_CAR has been set to null
MY_CAR = YOUR_CAR            -- true if sharing one car
MY_CAR.all = YOUR_CAR.all    -- true if the two cars are identical

I not in 1 .. 10                    -- range check
TODAY in WEEK_DAY                   -- subtype check
TODAY in DAY range MON .. FRI       -- same subtype check
0.1 = FRAC(0.2)                     -- qualification necessary to define the type
```

4.5.3 Adding Operators

The adding operators + and - return a result of the same type as the operands.

Operator	*Operation*	*Operand Type*	*Result Type*
+	addition	numeric type	same numeric type
-	subtraction	numeric type	same numeric type
&	catenation	one dimensional array type	same array type

For real types, the accuracy of the result is the accuracy of the operand type.

The adding operator & (catenation) is applied to two operands of an array type which has been declared to be one dimensional and whose index is specified by a type mark. The result is an array of the same type. (Note that an expression of the component type is regarded as a one component array of this type). For strings, this operation results in conventional string catenation.

For all numeric types, the exception RANGE_ERROR is raised if the result value is outside the range of the result type. The exception OVERFLOW is raised if the result value is beyond the implemented limits.

Examples:

```
Z + 0.1          -- addition is that of the type of Z, which must be real
"A" & "BCD"      -- catenation of a character with a string
```

4.5.4 Unary Operators

Unary operators are applied to a single operand and return a result of the same type.

Operator	*Operation*	*Operand Type*	*Result Type*
+	identity	numeric type	same numeric type
-	negation	numeric type	same numeric type
not	logical negation	BOOLEAN boolean array type	BOOLEAN same array type

The operator **not** can also be applied to arrays of boolean values on a component by component basis, just as for logical operators.

The exceptions RANGE_ERROR and OVERFLOW can be raised by the negation operation, just as for the subtraction operation.

4.5.5 Multiplying Operators

The operators * and / for integer and floating point values and the operator **mod** for integer values return a result of the same type as the operands.

Operator	*Operation*	*Operand Type*	*Result Type*
*	multiplication	integer floating	same integer type same floating type
/	integer division floating division	integer floating	same integer type same floating type
mod	modulus	integer	same integer type

Integer division and modulus are defined by the relation

A = (A/B)*B + (A **mod** B)

where (A **mod** B) has the sign of A and an absolute value less than the absolute value of B. Integer division satisfies the identity

(-A)/B = -(A/B) = A/(-B)

For fixed point values, the following multiplication and division operations are provided. The types of the left and right operands are denoted by L and R.

Operator	*Operation*	*Operand Type* L	R	*Result Type*
*	multiplication	fixed	integer	same as L
		integer	fixed	same as R
		fixed	fixed	*universal_fixed*
/	division	fixed	integer	same as L
		fixed	fixed	*universal_fixed*

Integer multiplication of fixed point values is equivalent to repeated addition and hence is an accurate operation. Division of a fixed point value by an integer does not involve a change in type but is approximate.

Fixed point multiplication may yield a value of an arbitrary accuracy (denoted by *universal_fixed* in the table). The result must be qualified (see 4.6) to ensure that the accuracy of the computation is explicitly controlled. The same considerations apply to division of a fixed point value by another fixed point value.

All multiplying operations can raise the exceptions RANGE_ERROR, OVERFLOW, or UNDERFLOW. The operations / and **mod** give the exception DIVIDE_ERROR when the right operand is zero.

Examples:

```
I   : INTEGER := 1;
J   : INTEGER := 2;
K   : INTEGER := 3;

X   : MY_FLOAT digits 6 := 1.0;
Y   : MY_FLOAT digits 6 := 2.0;

F   : FRAC delta 0.0001 := 0.1;
G   : FRAC delta 0.0001 := 0.1;
```

Expression	*Value*	*Result Type*
I*J	2	same as I and J, i.e. INTEGER
K/J	1	same as K and J, i.e. INTEGER
K **mod** J	1	same as K and J, i.e. INTEGER
X/Y	0.5	same as X and Y, i.e. MY_FLOAT
F/2	0.05	same as F, i.e. FRAC
3*F	0.3	same as F, i.e. FRAC
F*G	0.01	*universal_fixed*, qualification needed
FRAC(F*G)	0.01	given by qualification, i.e. FRAC
MY_FLOAT(J)*Y	4.0	MY_FLOAT, qualification of J converts its value to FLOAT

4.5.6 Exponentiating Operator

Operator	*Operation*	*Operand* L	*Type* R	*Result Type*
**	exponentiation	integer	positive integer	same as L
		floating	integer	same as L

Exponentiation of an operand by a positive exponent is equivalent to repeated multiplication (as indicated by the exponent) of the operand by itself. For a floating operand, the exponent can be negative, in which case the value is the reciprocal of the value with the positive exponent. This operation can raise the OVERFLOW, DIVIDE_ERROR, or RANGE_ERROR exception.

4.6 Qualified Expressions

A qualified expression is used to state the type of an expression explicitly, to constrain an expression to a given subtype, or, if neither case applies, to convert an expression to another type.

```
qualified_expression ::=
   type_mark(expression) | type_mark aggregate
```

4.6.1 Explicit Type or Subtype Specification

The same literal may appear in several types; it is then said to be overloaded. In these cases and whenever the type of a literal or aggregate is not known from the context, a qualified expression must be used to state the type explicitly.

In particular, an overloaded literal must be qualified in a subprogram call to an overloaded subprogram that cannot be identified on the basis of remaining parameter or result types, in a relational expression where both operands are overloaded literals, or in an array or loop parameter range where both bounds are overloaded enumeration literals.

Explicit type specification is also used to specify the result type of fixed point multiplication and division, to specify which one of a set of overloaded parameterless functions is meant, or to constrain a value to a given subtype.

Examples:

```
type MASKING_CODE  is (FIX, DEC, EXP, SIGNIF);
type INSTR_CODE    is (FIX, CLA, DEC, TNZ, SUB);

PRINT (MASKING_CODE(DEC));     -- DEC is of type MASKING_CODE
PRINT (INSTR_CODE(DEC));       -- DEC is of type INSTR_CODE

I in INSTR_CODE(FIX) .. INSTR_CODE(DEC)   -- qualification needed
I in INSTR_CODE range FIX .. DEC          -- qualification given by the context
```

4.6.2 Type Conversions

For numeric expressions, a qualified expression may specify a numeric type that is different from the type of the expression. In this case, the value of the expression is converted to the named type. With conversions involving real types, the converted value is within the accuracy of the specified type.

Examples of numeric type conversion:

```
REAL(2*I)          -- value is converted to floating point
INTEGER(1.6)       -- value is 2
INTEGER(-0.4)      -- value is 0
```

Explicit conversion is allowed between objects of derived types. The conversion may result in a change of representation, as described in chapter 13. Explicit conversion is also allowed between array types if the index types for each dimension are the same or derived from each other and if the component types are the same or derived from each other. Conversion involving an access type relates to the type of the accessed objects.

Example of conversion between derived types:

```
type A_FORM is new B_FORM;

X  : A_FORM;
Y  : B_FORM;

X := A_FORM(Y);
```

4.7 Allocators

An allocator specifies the dynamic creation of an object and the generation of an access value that designates the object.

allocator ::= **new** qualified_expression

The object created by the allocator is initialized with the value of the expression, which is qualified by the name of the access type.

Examples:

```
ELEMENT  := new LIST_ITEM(VALUE => 0, SUCC => null, PRED => null);
DOUBLE   := new PERSON(ME.all);
```

4.8 Static Expressions

A static expression is one whose value does not depend on any dynamically computed values of variables. Whenever the semantics require static expressions for the definition of some construct, these expressions are evaluated at compilation time and they must contain only the following:

(a) literals

(b) aggregates whose components are static expressions

(c) constants initialized by static expressions

(d) predefined operators, functions, and attributes

(e) qualified static expressions

(f) indexed and selected components of constants

5. Statements

Statements cause actions to be performed when executed. A statement may be simple or compound. A simple statement contains no other statement. A compound statement may contain simple statements and other compound statements.

```
sequence_of_statements ::= {statement}

statement ::=
      simple_statement | compound_statement
    | <<identifier>> statement

simple_statement ::=
      assignment_statement  | subprogram_call_statement
    | exit_statement        | return_statement
    | goto_statement        | assert_statement
    | initiate_statement    | delay_statement
    | raise_statement       | abort_statement
    | code_statement        | null;

compound_statement ::=
      if_statement      | case_statement
    | loop_statement    | accept_statement
    | select_statement  | block
```

A statement may be labeled, with an identifier enclosed by double angle brackets, e.g. <<HERE>>. Labels are used in exit and goto statements. Within the sequence of statements of a subprogram or module body, different labels must have different identifiers.

Execution of a null statement has no other effect. Blocks are described in the next chapter. Initiate, delay, abort, accept, and select statements are described in chapter 9 (Tasks). Raise statements are described in chapter 11 (Exceptions). Code statements are described in section 13.8 (Machine Code Insertions). The remaining statements are described here.

The statements in a sequence of statements are executed in succession unless an exception is raised or unless an exit, return, or goto statement is executed.

5.1 Assignment Statements

An assignment statement replaces the current value of a variable with a new value specified by an expression.

```
assignment_statement ::=    variable := expression;
```

The variable and the expression must be of the same type and the value of the expression must be compatible with any range, index, or discriminant constraint applicable to the variable. If the constraints are not checked during compilation, an execution time check is performed and raises an exception if it fails (the check may be omitted if the corresponding exception is suppressed, see 11.6).

Examples:

```
KEY_VALUE   := MAX_VALUE - 1;
SHADE       := BLUE;
```

Examples of constraints:

```
I, J : INTEGER range 1 .. 10;
K    : INTEGER range 1 .. 20;

I   := J;    -- identical ranges
K   := J;    -- compatible ranges
J   := K;    -- can only be checked during execution
             -- and may raise the RANGE_ERROR exception
```

5.1.1 Array and Slice Assignments

For an assignment to an array or to an array slice variable, the expression must denote a value with the same number of components. For slice assignments where the slice value refers to the same array as the slice variable, overlapping of index ranges is forbidden and raises the exception OVERLAP_ERROR.

Examples:

```
A  : STRING(0 .. 30);
B  : STRING(1 .. 31);

A := B;  -- same number of elements

A(1 .. 10) := A(11 .. 20);  -- non overlapping ranges
A(1 .. 5)  := "JAMES";      -- same number of elements
```

5.1.2 Record Assignments

For an assignment to a record variable declared with a specified discriminant value, the assigned record value must have the prescribed discriminant value. The discriminant of a record denoted by an access variable cannot be altered, not even by a complete record assignment.

Examples:

```
DISK_1, DISK_2 : PERIPHERAL(UNIT => DISK);

DISK_1 := (STATUS => OPEN, UNIT => DISK, CYLINDER => 1, TRACK => 1);
DISK_2 := DISK_1;
```

5.2 Subprogram Calls

A subprogram call invokes execution of a subprogram body. The call specifies the association of any actual parameters with formal parameters of the subprogram. An actual parameter is either a variable or the value of an expression.

```
subprogram_call_statement ::= subprogram_call;

subprogram_call ::=
   subprogram_name [(parameter_association {, parameter_association})]

parameter_association ::=
      [formal_parameter :=] actual_parameter
   | [formal_parameter =:] actual_parameter
   | [formal_parameter :=:] actual_parameter

formal_parameter ::= identifier

actual_parameter ::= expression
```

Actual parameters may be passed in positional order (positional parameters) or by explicitly naming the corresponding formal parameters (named parameters). For positional parameters, the actual parameter corresponds to the formal parameter with the same position in the formal parameter list. For named parameters, the corresponding formal parameter is explicitly given in the call. Named parameters may be given in any order.

Positional parameters and named parameters may be used in the same call provided that positional parameters occur first at their normal position, i.e. once a named parameter is used, the rest of the call must use only named parameters.

Examples:

```
RIGHT_SHIFT;
TABLE_MANAGER.INSERT(E);

SEARCH_STRING(STRING, CURRENT_POSITION, NEW_POSITION);

PRINT_HEADER(PAGES := 128, HEADER := TITLE, CENTER := TRUE);

REORDER_KEYS(NUM_OF_ITEMS, KEY_ARRAY :=: RESULT_TABLE);
```

5.2.1 Actual Parameter Associations

There are three forms for specifying actual parameters

(a) Input parameter association

[formal_parameter :=] actual_parameter

The corresponding formal parameter must have the mode **in**. Its value is provided by the actual parameter.

(b) Output parameter association

[formal_parameter =:] actual_parameter

The corresponding formal parameter must have the mode **out**. Its value is assigned to the actual parameter as a result of the execution of the subprogram.

(c) Input-output parameter association

[formal_parameter :=:] actual_parameter

The corresponding formal parameter must have the mode **in out**. Within the subprogram, the formal parameter permits access and assignment to the corresponding actual parameter.

An expression used as an **in** parameter is evaluated before the call. An expression used as an **out** or **in out** actual parameter must be a variable or a qualified variable. The identity of a variable **out** or **in out** actual parameter which is a selected component or an indexed component is established before the call.

5.2.2 Omission of Actual Parameters

An **in** parameter may be omitted from the actual parameters if the subprogram declaration specifies a default value for the corresponding formal parameter. In such cases, any remaining actual parameters must be named.

Example of procedure with default values:

```
procedure ACTIVATE( PROCESS  : in PROCESS_NAME;
                    AFTER    : in PROCESS_NAME := NO_PROCESS;
                    WAIT     : in TIME := 0.0;
                    PRIOR    : in BOOLEAN := FALSE);
```

Examples of its call:

```
ACTIVATE(X);
ACTIVATE(X, AFTER  := Y);
ACTIVATE(X, WAIT   := 5.0*SECONDS, PRIOR := TRUE);
```

5.2.3 Restrictions on Subprogram Calls

The type and constraint of each actual parameter must be consistent with those of the corresponding formal parameter, as for assignment. To prevent aliasing (i.e. multiple access to the same variable), a variable which is used as an actual **out** or **in out** parameter may not be used as another parameter of the same call. For this rule, any variable that is not local to the subprogram body is considered as an implicit **in** parameter if its value is read, and is considered as an **in out** parameter if it is directly or indirectly updated as a result of the call.

5.3 Return Statements

A return statement terminates execution of a subprogram.

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

A return statement can only appear in the sequence of statements of a subprogram. A return statement must not appear in an accept statement. For functions or value returning procedures, a return statement must include an expression whose value is the result of the subprogram.

Examples:

```
return;
return KEY_VALUE(LAST_INDEX);
```

5.4 If Statements

An if statement effects the choice of a sequence of statements based on the truth value of one or more conditions. The expressions appearing in conditions must be of the predefined type BOOLEAN.

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

condition ::=
       expression {and then expression}
   |   expression {or else expression}
```

Execution of an if statement results in evaluation of the conditions, one after the other (treating a final **else** as **elsif** TRUE **then**), until one evaluates to TRUE; then the corresponding sequence of statements is executed. If none of the conditions evaluates to TRUE, none of the sequences of statements is executed.

Examples:

```
if MONTH = DECEMBER and DAY = 31 then
   MONTH := JANUARY;
   DAY   := 1;
   YEAR  := YEAR + 1;
end if;

if INDENT then
   CHECK_LEFT_MARGIN;
   LEFT_SHIFT;
elsif UNDENT then
   RIGHT_SHIFT;
else
   CARRIAGE_RETURN;
   CONTINUE_SCAN;
end if;

if MY_CAR.OWNER.VEHICLE /= MY_CAR then
   FAIL ("INCORRECT RECORD");
end if;
```

5.4.1 Short Circuit Conditions

A condition may appear as a sequence of boolean expressions separated by **and then**. In such a case, evaluation of the constituent expressions proceeds in textual order until one evaluates to FALSE, in which case the value of the condition is FALSE; the condition is true only if all expressions evaluate to TRUE. Similarly, for expressions separated by **or else**, evaluation stops as soon as an expression evaluates to TRUE, in which case the value of the condition is TRUE; the condition is false only if all expressions evaluate to FALSE.

Examples:

```
if  MY_CAR.OWNER /= null  and then  MY_CAR.OWNER.AGE < 18  then
   MINOR := TRUE;
end if;

if  I = 0  or else  A(I) = HIT_VALUE  then
   return;
end if;
```

5.5 Case Statements

A case statement selects and executes one of several alternative sequences of statements. The selection is based on the value of an expression, of a discrete type, given at the head of the case statement.

```
case_statement ::=
   case expression of
      {when choice {| choice} => sequence_of_statements}
   end case;
```

Each alternative is preceded by a list of choices specifying the values for which the alternative is executed. Choices given in case statements follow the same rules as choices given in component associations for array aggregates (see 3.6.2). Thus, each possible value of the type or subtype of the expression must be given for one and only one alternative; the choice **others** can be given as the choice for the last alternative to cover all values not given in previous choices. Note that it is always possible to use a qualified expression to limit the number of choices that need be given explicitly.

Examples:

```
case SENSOR of
   when ELEVATION  => RECORD_ELEVATION (SENSOR_VALUE);
   when AZIMUTH    => RECORD_AZIMUTH   (SENSOR_VALUE);
   when DISTANCE   => RECORD_DISTANCE  (SENSOR_VALUE);
   when others     => null;
end case;

case TODAY of
   when MON         => COMPUTE_INITIAL_BALANCE;
   when FRI         => COMPUTE_CLOSING_BALANCE;
   when TUE .. THU  => GENERATE_REPORT(TODAY);
   when SAT .. SUN  => null;
end case;

case BIN_NUMBER((I mod 4) + 1) of
   when 1 => UPDATE_BIN(1);
   when 2 => UPDATE_BIN(2);
   when 3 | 4 =>
      EMPTY_BIN(1);
      EMPTY_BIN(2);
end case;
```

5.6 Loop Statements

A loop statement specifies that a sequence of statements in a basic loop is to be executed repeatedly zero or more times. Execution is terminated either when the iteration specification of the loop is exhausted or when an exit statement within the basic loop is executed.

```
loop_statement ::= [iteration_specification] basic_loop

basic_loop ::=
   loop
      sequence_of_statements
   end loop [identifier];

iteration_specification ::=
      for loop_parameter in [reverse] discrete_range
   |  while condition

loop_parameter ::= identifier
```

In a loop statement with a while clause, the condition is evaluated and tested before each execution of the basic loop. If the while condition is TRUE the loop is executed, if FALSE the loop statement is terminated.

In a loop statement with a for clause, the discrete range is evaluated only once, before execution of the loop statement. The loop parameter is implicitly declared as a local variable whose type is that of the elements in the discrete range. On successive loop iterations, the loop parameter is successively assigned values from the specified range. The values are assigned in increasing order unless the reserved word **reverse** is present, in which case the values are assigned in decreasing order.

If the range of a for loop is empty, the basic loop is not executed. Within the basic loop, the loop parameter acts as a constant. Hence the loop parameter may not be changed by an assignment statement, nor may it be given as an **out** or **in out** parameter of a subprogram call.

If a loop is a labeled statement, the label identifier must be repeated at the end of the loop after the reserved words **end loop**.

Examples:

```
while BID(I).PRICE < CUT_OFF.PRICE loop
   RECORD_BID(BID(I).PRICE);
   I := I + 1;
end loop;

while NEXT /= HEAD loop
   SUM  := SUM + NEXT.VALUE;
   NEXT := NEXT.SUCC;
end loop;

for I in BUFFER'FIRST .. BUFFER'LAST loop  -- valid even with empty range
   if BUFFER(I) /= " " then
      PUT(BUFFER(I));
   end if;
end loop;
```

5.7 Exit Statements

An exit statement causes explicit termination of an enclosing loop.

exit_statement ::= **exit** [identifier] [**when** condition];

The loop exited is the innermost loop, unless the exit statement identifies the label of an enclosing loop, in which case the named loop is exited. The exit statement may contain a condition, in which case termination occurs only if its value is TRUE. An exit statement may only appear within a loop. An exit statement cannot transfer control out of a subprogram, module, accept statement, or exception handler.

Example:

```
for I in 1 .. MAX_NUM_ITEMS loop
   GET_NEW_ITEM(NEW_ITEM);
   MERGE_ITEM(NEW_ITEM, STORAGE_FILE);
   exit when NEW_ITEM = TERMINAL_ITEM;
end loop;

<<MAIN_CYCLE>>
   loop
      -- initial statements
      exit MAIN_CYCLE when FOUND;
      -- final statements
   end loop MAIN_CYCLE;
```

5.8 Goto Statements

The execution of a goto statement results in an explicit transfer of control to another statement.

goto_statement ::= **goto** identifier;

The statement to which control is transferred must be labeled with the same identifier. The designated statement and the goto statement must both be within the same subprogram, module, or accept statement.

A goto statement cannot transfer control from outside into a compound statement, block, subprogram, module, accept statement, or exception handler. It may transfer control from one of the sequences of statements of an if statement or a case statement to another.

A goto statement cannot transfer control out of a subprogram, module, accept statement, or exception handler.

Example:

```
<<COMPARE>>
  if A(I) < ELEMENT then
     if LEFT(I) /= 0 then
        I := LEFT(I);
        goto COMPARE;
     end if;
     -- some statements
  end if;
```

5.9 Assert Statement

An assert statement states that a condition must hold whenever control reaches that point in the program.

assert_statement ::= **assert** condition;

The execution of an assert statement causes the evaluation of the condition, and the exception ASSERT_ERROR is raised if the condition is false.

Execution of assert statements may be omitted when the exception ASSERT_ERROR is suppressed by a pragma (see 11.6).

6. Declarative Parts, Subprograms, and Blocks

A declarative part contains declarations and related information that apply over a region of program text.

A subprogram is an executable program unit that is invoked by a subprogram call. Its definition can be given in two parts: a subprogram declaration defining its calling convention, and a subprogram body defining its execution.

A block allows one to make declarations local to the sequence of statements where they are used, without introducing a procedure. A block may be viewed as an anonymous procedure implicitly called at the place of its definition.

6.1 Declarative Parts

Blocks, subprograms, and modules may contain declarative parts.

```
declarative_part ::=
   [use_clause] {declaration} {representation_specification} {body}

body ::= [visibility_restriction] unit_body | body_stub

unit_body ::= subprogram_body | module_specification | module_body
```

The successive constituents of a declarative part are elaborated in the order in which they appear in the program text. Expressions appearing in declarations or representation specifications (see 13) are evaluated during this elaboration. A subprogram must not be called within such an expression if the subprogram body appears later in the declarative part. In particular, these rules apply to formal parts of subprogram specifications and to constraints of objects, types, and subtypes.

The body of a subprogram or module declared in the declarative part of a block or subprogram must be provided in the same declarative part. The body of a subprogram or module declared in a module specification must be provided in the corresponding module body. If the body of such a unit is a separately compiled subunit (see 10.2) it must be represented by a body stub at the place where it would otherwise appear.

A declarative part can also contain a use clause (see 8.4).

6.2 Subprogram Declarations

A subprogram declaration specifies the designator of a subprogram, its nature (function or subprogram), its formal parameters (if any), and the type of any returned value.

```
subprogram_declaration ::=
      subprogram_specification;
   |  subprogram_nature designator is generic_instantiation;

subprogram_specification ::= [generic_clause]
   subprogram_nature designator [formal_part] [return type_mark [constraint]]

subprogram_nature ::= function | procedure

designator ::= identifier | character_string

formal_part ::= (parameter_declaration {; parameter_declaration})

parameter_declaration ::=
   identifier_list : mode type_mark [constraint] [:= expression]

mode ::= [in] | out | in out
```

A designator that is a character string is used in function declarations for overloading operators of the language. Such a string must denote one of the existing operator symbols (see 4.5).

A subprogram specification including a generic clause specifies a generic subprogram; an instance of such a generic subprogram is declared with a subprogram declaration including a generic instantiation (see 12).

A parameter declaration or constraint on the result cannot contain an identifier declared in another parameter declaration of the same formal part.

Examples of subprogram declarations:

```
procedure TRAVERSE_TREE;
procedure RIGHT_INDENT(MARGIN : out LINE_POSITION);
procedure INCREMENT(X : in out INTEGER);
procedure RANDOM return REAL range -1.0 .. 1.0;

function COMMON_PRIME (M, N  : INTEGER) return INTEGER;
function DOT_PRODUCT   (X, Y : VECTOR)  return REAL;
function "*" (X, Y : MATRIX) return MATRIX;
```

Notes:

All subprograms can be called recursively and are reentrant.

6.3 Formal Parameters

The formal parameters of a subprogram are considered local to the subprogram. A parameter has one of three modes:

in The parameter acts as a local constant whose value is provided by the corresponding actual parameter.

out The parameter acts as a local variable whose value is assigned to the corresponding actual parameter as a result of the execution of the subprogram.

in out The parameter acts as a local variable and permits access and assignment to the corresponding actual parameter.

If no mode is explicitly given, the mode **in** is assumed. The components of **in** parameters that are arrays, records, or objects denoted by access values must not be changed by the subprogram.

For **in** parameters, the parameter declaration may also include a specification of a default expression, whose value is implicitly assigned to the parameter if no explicit value is given in the call. This expression is evaluated when the subprogram specification is elaborated.

For all modes, access to the actual parameters can be provided either throughout the execution of the subprogram body or by copying the corresponding actual parameter before the call (**in** parameters), after the call (**out** parameters) or both (**in out** parameters). The effect of a subprogram that is abnormally terminated by the occurrence of an exception is undefined; its actual **in out** and **out** parameters may or may not have been updated.

In the absence of aliasing (see 5.2.3) the effect of a subprogram call is the same whether or not copying is used for parameter passing, unless the subprogram execution is abnormally terminated. A program that relies on some assumption regarding the actual mechanism used for parameter passing is therefore erroneous.

Examples of in parameters with default values:

```
procedure PRINT_HEADER(PAGES   : in INTEGER;
                       HEADER  : in LINE := BLANK_LINE;
                       CENTER  : in BOOLEAN := TRUE);

procedure ACTIVATE( PROCESS  : in PROCESS_NAME;
                    AFTER    : in PROCESS_NAME := NO_PROCESS;
                    WAIT     : in REAL := 0.0;
                    PRIOR    : in BOOLEAN := FALSE);
```

6.4 Subprogram Bodies

A subprogram body specifies the execution of a subprogram.

```
subprogram_body ::=
    subprogram_specification is
      declarative_part
    begin
      sequence_of_statements
    [ exception
      {exception_handler}]
    end [designator];
```

The subprogram specification provided in a subprogram body must be identical to the specification given in the corresponding subprogram declaration, if both are given. A subprogram declaration must be given if the subprogram is defined in the visible part of a module, or if it is called by other subprogram or module bodies that appear before its own body. Otherwise, it can be omitted and the specification appearing in the body acts as a subprogram declaration. The elaboration of a subprogram body consists of the elaboration of its specification unless the latter elaboration has already been done.

Upon each call to a subprogram, the association between actual and formal parameters is established (see 5.2), the declarative part of the body is elaborated, and the statements of the body are executed. Upon completion of the body, assignment to **out** and **in out** actual parameters is completed, if necessary (see 6.3), and then return is made to the caller. A subprogram body may contain exception handlers to service exceptions occurring during its execution (see 11).

The optional designator at the end of the subprogram body must repeat the designator of the subprogram specification.

Example of subprogram body:

```
procedure PUSH(E : in ELEMENT_TYPE; S : in out STACK) is
begin
   if S.INDEX = S.SIZE then
      raise STACK_OVERFLOW;
   else
      S.INDEX := S.INDEX + 1;
      S.SPACE(S.INDEX) := E;
   end if;
end PUSH;
```

Notes:

A subprogram body may be expanded in line at each call if its declarative part includes the declarative pragma:

```
pragma INLINE;
```

The meaning of a subprogram is not changed by the pragma INLINE, which is merely a recommendation to the compiler. Thus, an inline subprogram could be recursive or separately compiled.

6.5 Function Subprograms

A function is a subprogram that computes a value. A function can only have **in** parameters and must contain a return clause specifying the type of its returned value. The statement list in the function body must include one or more return statements specifying the returned value. An attempt to leave a function otherwise than by a return statement (i.e. by reaching the final **end**) causes a NO_VALUE_ERROR exception to be raised.

Side effects, e.g. assignments to non-local variables, are not allowed within functions, whether directly, or indirectly through other subprogram calls. Hence, if function calls occur in expressions, they can be rearranged in any order consistent with the properties of the operators.

If a parameter belongs to an access type, the parameter must be viewed as providing access to the complete collection of dynamically allocated objects. For functions, this collection is considered as an implicit **in** parameter. As a consequence, within the function body there can be no alteration to any object designated by such a parameter or designated by a local variable of the access type. Similarly, allocators cannot appear in a function body.

Value returning procedures obey rules similar to those of functions: a value returning procedure can only have **in** parameters, its declaration must contain a return clause, and its body may only be left by a return statement. However, assignments to global variables are permitted within value returning procedures. Calls of such procedures are only valid at points of the program where the corresponding variables are not within the scope of their declaration. The order of evaluation of these calls is strictly that given in the text of the program. Calls to value returning procedures are only allowed in expressions appearing in assignment statements, initializations, and procedure calls.

Examples:

```
function DOT_PRODUCT(X, Y : VECTOR) return REAL is
   SUM : REAL := 0.0;
begin
   assert X'FIRST  = Y'FIRST;
   assert X'LAST   = Y'LAST;
   for I in X'FIRST .. X'LAST loop
      SUM := SUM + X(I)*Y(I);
   end loop;
   return SUM;
end DOT_PRODUCT;

package UNIQUE_NUMBER_GENERATOR is
   procedure GENERATOR return INTEGER;  -- value returning procedure
end;

package body UNIQUE_NUMBER_GENERATOR is
   COUNT : INTEGER := 0;      -- COUNT is not visible where GENERATOR is called
   procedure GENERATOR return INTEGER is
   begin
      COUNT := COUNT + 1;    -- side effect on COUNT
      return COUNT;
   end GENERATOR;
end UNIQUE_NUMBER_GENERATOR;
```

6.6 Overloading of Subprograms

The same subprogram designator can be given in several otherwise different subprogram specifications; it is then said to be *overloaded*. The declaration of an overloaded subprogram does not hide a previous subprogram declaration unless the order, names, modes, and types of the parameters, and the result type, if any, are identical in both declarations (a default expression for an **in** parameter is ignored here). Such redefinition is, of course, illegal within the same declarative part. Overloaded definitions may, but need not, occur in the same declarative part.

A call to an overloaded subprogram is ambiguous (and therefore illegal) if the type, mode, and name information derived from the actual parameter associations and the type information required for the result are not sufficient to identify exactly one overloaded specification. Ambiguities may be resolved by the use of a qualified expression, or by the naming of parameters.

Examples of overloaded subprograms:

```
procedure PUT(X : INTEGER);
procedure PUT(X : STRING);

procedure CHANGE(C : COLOR);
procedure CHANGE(L : LIGHT);
procedure CHANGE(F : LIGHT);
```

Example of calls:

```
PUT(28);
PUT("no possible ambiguity here");

CHANGE(COLOR(RED));
CHANGE(C := RED);
CHANGE(F := RED);
-- CHANGE(RED) would be ambiguous since RED may denote a value of either COLOR or LI

X + 1.5  -- the floating or fixed point type of X identifies the relevant "+"
```

6.6.1 Overloading of Operators

A function named by a character string is used to define an additional meaning for an operator. The overloading of operators is identical to overloading of other subprograms, except that the character string must denote one of the operators in the language.

Overloading is permitted for both unary and binary operators. A unary operator can only be overloaded as a unary and a binary as a binary. Overloading does not change the precedence of an operator. An overloading of a relational operator must have the result type BOOLEAN. The operator /= must not be overloaded explicitly, since every overloading of the operator = results in an implicit overloading of /=.

Examples:

```
function "*" (X, Y : MATRIX)  return MATRIX;
function "*" (X, Y : VECTOR)  return VECTOR;
```

Notes:

Good usage of operator overloading should preserve their mathematical properties.

6.7 Blocks

A block introduces a sequence of statements, optionally preceded by a governing declarative part.

```
block ::=
   [ declare
      declarative_part]
    begin
      sequence_of_statements
   [ exception
      {exception_handler}]
    end [identifier];
```

Execution of a block results in the elaboration of its declarative part followed by the execution of the sequence of statements. A block may also contain exception handlers to service exceptions occurring in the block (see 11). If a block is labeled, the optional identifier appearing at the end of the block must repeat the label.

Example of a labeled block:

```
<<SWAP>>
  declare
    TEMP : INTEGER;
  begin
    TEMP := V; V := U; U := TEMP;
  end SWAP;
```

7. Modules

A program can be composed of program units of three kinds. These are subprograms and two forms of modules, package modules and task modules. This chapter describes the common properties of package and task modules and the few specific properties of package modules. The specific properties of task modules are described in Chapter 9.

Modules allow the specification of groups of logically related entities. In their simplest form modules can represent pools of common data and type declarations. In addition, modules can be used to describe groups of related subprograms and encapsulated data types, whose inner workings are concealed and protected from their users.

7.1 Module Structure

A module is generally provided in two parts: a module specification and a module body with the same identifier. The simplest forms of modules, those representing pools of data and types, do not require a module body.

```
module_declaration ::=
      [visibility_restriction] module_specification
   |  module_nature identifier [(discrete_range)] is generic_instantiation;

module_specification ::=
   [ generic_clause]
     module_nature identifier [(discrete_range)] [is
       declarative_part
   [ private
       declarative_part]
     end [identifier]];

module_nature ::= package | task

module_body ::=
     module_nature body identifier is
       declarative_part
   [ begin
       sequence_of_statements]
   [ exception
       [exception_handler]]
     end [identifier];
```

A module specification contains a declarative part called the visible part and an optional declarative part called the private part. Elaboration of a module declaration results in the elaboration of these declarative parts and therefore in the allocation of the variables of the module specification and the assignment of any initial values.

A module specification with a generic clause defines a generic module. Instances of generic modules can be obtained by module declarations including a generic instantiation (see 12).

A module declaration may include a discrete range after the identifier. This only applies to task modules and the identifier then denotes a family of tasks.

The elaboration of a task body has no other effect. The elaboration of its declarative part and the execution of its sequence of statements is caused by the execution of an initiate statement (see 9.3). The elaboration of a package body causes the elaboration of its declarative part and the execution of the sequence of statements, if any. These statements can be used to achieve further initializations.

Module bodies and the visible parts of packages may contain further module declarations. The body of any unit declared in a module specification must appear in the corresponding module body.

7.2 Module Specifications

The first declarative part of a module specification is called its visible part. The entities declared in the visible part can be made visible to other program units by means of a use clause (see 8.4) or selected components (see 4.1.2). A module consisting of only a module specification (i.e., without a module body) can be used to represent a group of common constants or variables, or a common pool of data and types.

Example of a group of common variables:

```
package PLOTTING_DATA is
   PEN_UP : BOOLEAN;

   CONVERSION_FACTOR,
   X_OFFSET, Y_OFFSET,
   X_MIN, X_MAX,
   Y_MIN, Y_MAX : REAL;

   X_VALUE, Y_VALUE : array (1 .. 500) of REAL;
end PLOTTING_DATA;
```

Example of common pool of data and types:

```
package WORK_DATA is
   type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
   type DURATION   is delta 0.01 range 0.0 .. 24.0;
   type TIME_TABLE is array (MON .. SUN) of DURATION;

   WORK_HOURS   : TIME_TABLE;
   NORMAL_HOURS : constant TIME_TABLE :=
                     (MON .. THU => 8.25, FRI => 7.0, SAT | SUN => 0.0);
end WORK_DATA;
```

The visible part contains all the information that another program unit is able to know about the module.

7.3 Module Bodies

The visible part of a module may contain the specification of subprograms or the specification of other modules. In such cases, the bodies of the specified units must appear within the declarative part of the module body. This declarative part can also include local declarations and local program units needed to implement the visible items.

In contrast to the entities declared in the visible part, the entities declared in the module body are not accessible outside the module. As a consequence, a module with a module body can be used for the construction of a group of related subprograms (a *package* in the usual sense), where the logical operations accessible to the user are clearly isolated from the internal entities.

Example of a package:

```
package RATIONAL_NUMBERS is
   type RATIONAL is
      record
         NUMERATOR     : INTEGER;
         DENOMINATOR   : INTEGER range 1 .. INTEGER'LAST;
      end record;

   function "="  (X,Y : RATIONAL) return BOOLEAN;
   function "+"  (X,Y : RATIONAL) return RATIONAL;
   function "*"  (X,Y : RATIONAL) return RATIONAL;
   -- Note: "=" hides predefined equality for RATIONAL operands
end;

package body RATIONAL_NUMBERS is

   procedure SAME_DENOMINATOR (X,Y : in out RATIONAL) is
   begin
      -- reduces X and Y to the same denominator
   end;

   function "=" (X,Y : RATIONAL) return BOOLEAN is
      U,V : RATIONAL;
   begin
      U := X;
      V := Y;
      SAME_DENOMINATOR (U,V);
      return U.NUMERATOR = V.NUMERATOR;
   end "=";

   function "+"  (X,Y : RATIONAL) return RATIONAL is    ... end "+";
   function "*"  (X,Y : RATIONAL) return RATIONAL is    ... end "*";

end RATIONAL_NUMBERS;
```

Notes:

A variable declared in a module specification or body retains its value between calls to subprograms declared in the visible part. Such a variable is said to be *own* to the module.

7.4 Private Type Declarations

The structural details of some declared types may be irrelevant to their logical use outside a module. This may be accomplished by providing a private type declaration.

```
private_type_declaration ::=
   [restricted] type identifier is private;
```

A private type declaration can only appear in the visible part of a module. The full declaration of the private type must appear in the private part of the module specification. Such types are called private types.

For a private type (not designated as restricted), the only information available to other program units is that given in the visible part of the defining module. Hence, the name of the type and the operations specified in this visible part are available. In addition, assignment and the predefined comparison for equality or inequality are available (unless a redefinition of equality hides the predefined equality and, as a consequence, also redefines inequality).

These are the only externally available operations on objects of a private type. External units can declare objects of the private type and apply available operations to the objects. In contrast, external units cannot access the structural details of objects of private types directly.

A constant value of a private type can be declared in the visible part as a deferred constant. Its actual value must be specified in the private part by redeclaring the constant in full.

Assignment and the predefined comparison for equality or inequality are not available for private type declarations containing the reserved word **restricted**. Thus if a type is restricted, the only operations available on objects of the type are those defined by the subprograms declared in the visible part. A user can of course define subprograms calling the visible operations.

Example:

In the next example a private type KEY is defined which only has the operations of assignment, comparison for equality or inequality, comparison for "<", and an operation to create a value of type KEY.

```
package KEY_MANAGER is
   type KEY is private;
   NULL_KEY : constant KEY;
   procedure GET_KEY (K : out KEY);
   function "<" (X, Y : KEY) return BOOLEAN;
private
   type KEY is new INTEGER range 0 .. INTEGER'LAST;
   NULL_KEY : constant KEY := 0;
end;
```

```
package body KEY_MANAGER is
   LAST_KEY : KEY := 0;
   procedure GET_KEY(K : out KEY) is
   begin
      LAST_KEY := LAST_KEY + 1;
      K := LAST_KEY;
   end GET_KEY;

   function "<" (X, Y : KEY) return BOOLEAN is
   begin
      return INTEGER(X) < INTEGER(Y);
   end "<";
end KEY_MANAGER;
```

Notes:

The expression X<Y within the body of the function "<" would be a recursive call of "<". Hence, qualified expressions are necessary in the relation

```
INTEGER(X) < INTEGER(Y)
```

Example:

In the example below, an external subprogram making use of I_O_PACKAGE may obtain a file name by calling OPEN and later use it in calls to READ and WRITE. Thus, outside the module, a file name obtained from OPEN acts as a kind of password. Its internal properties (e.g., containing a numeric value) are not known and no other operations (such as addition or comparison of internal names) can be performed on a file name.

```
package I_O_PACKAGE is
   restricted type FILE_NAME is private;

   procedure OPEN  (F : in out FILE_NAME);
   procedure READ  (ITEM : out INTEGER; F : in FILE_NAME);
   procedure WRITE (ITEM : in  INTEGER; F : in FILE_NAME);
private
   type FILE_NAME is
      record
         INTERNAL_NAME : INTEGER := 0;
      end record;
end I_O_PACKAGE;

package body I_O_PACKAGE is
   LIMIT : constant INTEGER := 200;
   type FILE_DESCRIPTOR is record    ...    end record;
   DIRECTORY : array (1 .. LIMIT) of FILE_DESCRIPTOR;
   ...
   procedure OPEN  (F : in out FILE_NAME) is    ...    end;
   procedure READ  (ITEM : out  INTEGER;  F : in FILE_NAME) is    ...    end;
   procedure WRITE (ITEM : in   INTEGER;  F : in FILE_NAME) is    ...    end;
begin
   ...
end I_O_PACKAGE;
```

This example is characteristic of any case where complete control over the operation of a type is desired. Such packages serve a dual purpose. They prevent a user from making use of the internal structure of the type. They also implement the notion of an encapsulated data type where the only operations over the type are those given in the module.

7.5 An Illustrative Table Management Package

The following example illustrates the use of package modules in providing high level procedures with a simple interface to the user.

The problem is to define a table management package for inserting and retrieving items. The items are inserted into the table as they are posted. Each posted item has an order number. The items are retrieved according to their order number, where the item with the lowest order number is retrieved first.

From the user's point of view, the package is quite simple. There is a type called ITEM designating table items, a procedure INSERT for posting items, and a procedure RETRIEVE for obtaining the item with the lowest order number. There is a special item NULL_ITEM that is returned when the table is empty, and an exception TABLE_FULL that may be raised by INSERT.

A sketch of a module implementing such a package is given below. Only the visible part of the package is exposed to the user.

```
package TABLE_MANAGER is

  type ITEM is
    record
      ORDER_NUM   : INTEGER;
      ITEM_CODE   : INTEGER;
      ITEM_TYPE   : CHARACTER;
      QUANTITY    : INTEGER;
    end record;

  NULL_ITEM : constant ITEM :=
    (ORDER_NUM | ITEM_CODE | QUANTITY => 0, ITEM_TYPE => ' ');

  procedure INSERT   (NEW_ITEM   : in  ITEM);
  procedure RETRIEVE (FIRST_ITEM : out ITEM);

  TABLE_FULL : exception;  -- may be raised by INSERT
end;
```

The details of implementing such packages can be quite complex, in this case involving a two way linked table of internal items. A local housekeeping procedure EXCHANGE is used to move an internal item between the busy and the free lists. The initial table linkages are established by the initialization part. The package body need not be shown to the users of the package.

```
package body TABLE_MANAGER is
   SIZE : constant INTEGER := 2000;
   subtype INDEX is INTEGER range 0 .. SIZE;

   type INTERNAL_ITEM is
      record
         CONTENT  : ITEM;
         SUCC     : INDEX;
         PRED     : INDEX;
      end record;

   TABLE : array (INDEX'FIRST .. INDEX'LAST) of INTERNAL_ITEM;
   FIRST_BUSY_ITEM  : INDEX := 0;
   FIRST_FREE_ITEM  : INDEX := 1;

   function FREE_LIST_EMPTY  return BOOLEAN is ... end;
   function BUSY_LIST_EMPTY  return BOOLEAN is ... end;
   procedure EXCHANGE (FROM : in INDEX; TO : in INDEX) is ... end;

   procedure INSERT (NEW_ITEM : in ITEM) is
   begin
      if FREE_LIST_EMPTY then
         raise TABLE_FULL;
      end if;
      -- remaining code for INSERT
   end INSERT;

   procedure RETRIEVE (FIRST_ITEM : out ITEM) is ... end;

begin
   -- initialization of the table linkages
end TABLE_MANAGER;
```

8. Visibility Rules

This chapter describes the rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the text of the program. These rules are stated here as applying to identifiers. They apply equally to character strings used as function designators or enumeration literals.

A declaration associates an identifier with a program entity, such as a variable, a type, a subprogram, a formal parameter, a record component, etc. The region of text over which a declaration has an effect is called the *scope* of a declaration.

The same identifier can be introduced by different declarations in the text of a program and thus be associated with alternative entities. Hence the scopes of several declarations with the same identifier can overlap.

Overlapping scopes for declarations with the same identifier can occur because of overloading of subprograms or of enumeration literals (see 6.6 and 3.5.1). Overlapping scopes can also occur because of nesting. In particular, subprograms, modules, and blocks can be nested within each other; similarly these units can contain nested record type definitions or nested loop statements.

At a given point of text, the declaration of an entity with a certain identifier is said to be *visible* if this entity is an acceptable meaning for an occurrence of the identifier.

For overloaded identifiers, there can be *several* meanings acceptable at a given point, and the ambiguity must be resolved by the rules of overloading (see 4.6 and 6.6). For other identifiers (the usual case and the case considered in this chapter) there can be *at most one* acceptable meaning. By convention, an identifier is said to be visible if its declaration is visible. The *visibility rules* are the rules defining which identifiers are visible at various points of the text.

8.1 Scope of Declarations

Entities can be introduced by declarations in various ways. An entity can be declared in a declarative part of a block, subprogram, or module. An enumeration literal is declared by its occurrence in an enumeration type definition, a loop parameter by its occurrence in an iteration specification. Finally, entities can be declared as record components or as formal parameters of subprograms, entries, and generic clauses.

The scopes of these various forms of declarations and the scope of labels are defined as follows:

- The scope of a declaration given in the declarative part of a block, subprogram body, or module body extends from (and includes) the declaration up to the end of the corresponding block, subprogram, or module.

- The scope of a declaration given in the visible or private part of a module extends from (and includes) the declaration to the end of the module specification. It also extends over the corresponding module body.

- The scope of a declaration given in the visible part of a module also extends to the end of the scope of the module declaration itself.

- The scope of an enumeration literal is the scope of the enumeration type declaration (or definition) itself.

- The scope of a record component extends from the component declaration to the end of the scope of the record type declaration (or definition) itself.

- The scope of an (unnamed) enumeration or record type definition, itself given within a record type definition, extends to the end of the scope of the enclosing definition (or declaration).

- The scope of a formal parameter of a subprogram, entry, or generic clause extends from the parameter declaration to the end of the scope of the declaration of the subprogram, entry, or generic unit itself.

- The scope of a loop parameter extends to the end of the corresponding loop.

- The scope of a label extends from the first occurrence of a label to the end of the innermost enclosing compound statement, subprogram, or module. The first occurrence of a label can be either the label itself or its use in a goto statement.

8.2 Visibility of Identifiers

As defined in the previous section, the scope of a declaration always extends at least until the end of the language construct enclosing the declaration (either a block, a subprogram, an accept statement, a module, a record type definition, or a loop statement). In addition, the scope extends outside the enclosing construct for record components, formal parameters, and items declared in a module visible part.

The declaration of an identifier is visible at a given point of text if this point is within the construct enclosing its declaration, but not within an inner construct containing another declaration with the same identifier. An entity that is visible in this manner is *directly visible*, that is, it can be named simply by its identifier.

An entity declared in an enclosing construct is said to be *hidden* within an inner construct containing another declaration with the same identifier. A subprogram declaration hides another subprogram only if their specifications are equivalent with respect to the rules of subprogram overloading (see 6.6). An enumeration literal overloads but does not hide another enumeration literal. Redeclaration (as opposed to overloading) is not allowed within the same declaration list or component list.

The name of an entity declared immediately within a subprogram or module can always be written as a selected component within this unit, whether it is visible or hidden. The name of the unit (which must be visible) is then used as a prefix. Thus, component selection has the effect of opening the visibility for the occurrence of the identifier after the dot.

Outside its construct enclosing its declaration, but within its scope a record component, a formal parameter, or an item of a module visible part can be made visible as follows:

- A record component is made visible by a selected component whose prefix names a record of the corresponding type. It is also visible as a choice in an aggregate of the type.
- A formal parameter of a subprogram, entry, or generic clause is visible within named parameter associations of corresponding subprogram calls, entry calls, or generic instantiations.
- An entity declared within a module visible part is made visible by a selected component whose prefix names the module. It may also be made directly visible via a use clause (see 8.4).

Example:

```
procedure P is
   A  : BOOLEAN;
   B  : BOOLEAN;

   procedure Q is
      C  : BOOLEAN;
      B  : BOOLEAN;   -- an inner redeclaration of B
   begin
      ...
      B  := A;     -- means Q.B  := P.A;
      C  := P.B;   -- means Q.C  := P.B;
   end;
begin
   ...
   A := B;   -- means P.A := P.B;
end;
```

Restrictions on redeclarations:

An identifier used (as opposed to being declared) in one declaration in a declaration (or component) list may not be redeclared in subsequent declarations of the same list. Thus the following list of declarations is illegal:

```
M  : constant INTEGER := 2*N;   -- using N from outer scope
N  : constant INTEGER := 10;    -- illegal redeclaration of N
```

A variable or constant of an enumeration type cannot hide an enumeration value of the type. The same restriction applies to a parameterless function returning a result of an enumeration type.

Note on redeclaration:

An inner declaration of an object of a given type hides an outer declaration of a parameterless function with the same identifier and type, and vice versa.

8.3 Restricted Program Units

By means of a visibility restriction, a program unit may restrict the visibility it otherwise has of outer units.

visibility_restriction ::= **restricted** [visibility_list]

visibility_list ::= (*unit*_name {, *unit*_name})

In all cases, the predefined identifiers are visible within the restricted program unit. If no visibility list is given, no other identifiers are visible. If there is a visibility list, the first name can (but need not) be the name of a unit enclosing the restricted unit. Entities declared within the enclosing unit (if given) are visible as usual. Other names, if given, must be the names of modules that are outside the given enclosing unit or the restricted unit itself. These module names are also visible, and thus can be used in selected components and use clauses.

The outer modules could be library modules. A module body, whether restricted or not, always sees the visible part and the private part, if any, of its own module specification. Within a restricted program unit, a visibility restriction may be locally superseded by another visibility restriction given for an inner unit.

Example:

```
procedure MAIN is
   U : BOOLEAN;

   package A is
      LA : BOOLEAN;
   end;

   package B is
      LB : BOOLEAN;
   end;

   restricted(A)
   procedure OUTSIDE is
      V : BOOLEAN;

      restricted(OUTSIDE, INPUT_OUTPUT)
      procedure DISPLAY(W : BOOLEAN) is
      begin
         -- OUTSIDE, V, W, DISPLAY, and INPUT_OUTPUT are visible names.
         -- The identifiers of the visible part of the library module INPUT_OUTPUT
         -- can be denoted by selected components, or directly if a use clause is given
         -- for the module. The identifiers B, LB, A, LA, U, MAIN are not visible.
      end DISPLAY;
   begin
      -- A, OUTSIDE, V and DISPLAY are visible names
      -- The name A.LA is legal
      -- The name LA can be made directly visible by a use clause for A
      -- The identifiers B, LB, U, MAIN are not visible
   end OUTSIDE;
begin
   -- U, A, B, OUTSIDE are visible names
   -- The names A.LA and B.LB are legal
   -- The name LA and LB can be made directly visible by a use clause for A and B
end MAIN;
```

Notes:

If the visibility list includes the name of an enclosing unit, the names of modules local to this unit are already visible and hence must not be included in the visibility list.

8.4 Use Clauses

If the name of a module is visible at a given point of text, the identifiers declared within the visible part of the module can be denoted by selected components. In addition, these identifiers can be made directly visible by means of a use clause at the start of a declarative part.

use_clause ::= **use** *module*_name {, *module*_name};

The names appearing in the use clause must be visible module names.

In order to define the set of identifiers that are made visible by use clauses at a given point of the text, consider the set of module names appearing in the use clauses of the current and all enclosing units, up to the innermost enclosing restricted unit.

An identifier is made visible by a use clause if it is defined in the visible part of one and only one of these modules and if it is not visible otherwise.

Several overloaded identifiers (subprograms or enumeration literals) can be made visible by use clauses as long as none of them constitutes a redefinition of an otherwise visible identifier or of an identifier of another module in the set.

Thus an identifier made visible by a use clause can never hide another identifier although it may overload it. If an identifier appears in several used modules or is otherwise visible, the entity corresponding to its definition in one of the modules must still be denoted as a selected component. Renaming and subtype declarations may help avoiding excessive use of selected components.

Example 1:

```
procedure R is
   use TRAFFIC, WATER_COLORS;
   -- subtypes used to resolve the conflicting type name COLOR
   subtype T_COLOR  is TRAFFIC.COLOR;
   subtype W_COLOR  is WATER_COLORS.COLOR;

   SIGNAL : T_COLOR;
   PAINT  : W_COLOR;
begin
   SIGNAL := GREEN;   -- that of TRAFFIC
   PAINT  := GREEN;   -- that of WATER_COLORS
   ...
end R;
```

Example 2:

```
package D is
  T, U, V : BOOLEAN;
end D;

procedure P is

  package E is
    B, W, V : INTEGER;
  end E;

  procedure Q is
    T, X : REAL;
  begin
    ...
    declare
      use D, E;
    begin
      -- the name T means Q.T, not D.T
      -- the name U means D.U
      -- the name B means E.B
      -- the name W means E.W
      -- the name X means Q.X
      -- the name V is illegal : must be written either D.V or E.V
      ...
    end;
  end Q;
begin
  ...
end P;
```

8.5 Renaming

A renaming declaration associates a local name with an entity.

```
renaming_declaration ::=
      identifier  : type_mark   renames name;
    | identifier  : exception   renames name;
    | subprogram_nature designator renames [name.]designator;
    | module_nature identifier renames name;
```

The identity of the item following the reserved word **renames** is established when the renaming declaration is elaborated. The newly declared identifier (or designator) takes on the same properties (such as constancy, parameter types, and constraints, etc.) as the renamed entity.

A label cannot be renamed. An entry can only be renamed as a procedure. A subtype can effectively be used for renaming types as in

```
subtype ST is S.T;
```

Renaming may be used to resolve name conflicts (see example in section 8.4), to achieve partial evaluation and to act as a shorthand.

Examples:

```
procedure TMR   renames TABLE_MANAGER.RETRIEVE;
procedure SORT  renames QUICKSORT2;
task LC renames LINK_CONTROLLER(6);

declare
   L   : PERSON renames LEFTMOST_PERSON;
   R   : PERSON renames TO_BE_PROCESSED(NEXT);
begin
   L.AGE  := L.AGE  + 1;
   R.AGE  := R.AGE  + 1;
end;

FULL : exception renames TABLE_MANAGER.TABLE_FULL;
```

Notes:

Renaming does not hide the old name.

8.6 Predefined Environment

All predefined identifiers, for example built-in types, operators, and so forth, are assumed to be defined in the predefined module STANDARD given in Appendix C. Other installation defined modules may be included in the default environment by the pragma

```
pragma ENVIRONMENT (module_name {, module_name});
```

All identifiers declared in the visible part of the modules of the default environment are assumed declared at the outermost level of a program. Visibility restrictions do not affect the visibility of these predefined identifiers.

9. Tasks

Tasks are modules that may operate in parallel. Parallel tasks may be implemented on multicomputers, multiprocessors, or with interleaved execution on a single processor.

9.1 Task Declarations and Task Bodies

A task declaration is a module declaration whose module nature is the reserved word **task**. A task consists of two parts: the task specification and the task body. The specification can specify either a single task or a family of similar tasks whose individual members are denoted by an index from a discrete range. The task specification (like a package specification) comprises a visible part and an optional private part.

The visible part of a task specification consists of declarations specifying the interface between the task and other external units. Entry declarations are allowed in the visible part: an entry is used for communication between tasks in mutual exclusion. Declarations of variables and modules are not allowed in the visible part.

A task body specifies the execution of a task. The body can contain accept and select statements. It can also contain local entry declarations.

Examples of task declarations:

```
task PRODUCER_CONSUMER is
   entry READ (V  : out  ELEM);
   entry WRITE(E  : in   ELEM);
end PRODUCER_CONSUMER;

task MULTIPLEXER is
   type PRINTER is private;
   entry OPEN (P : out  PRINTER);
   entry CLOSE(P : in   PRINTER);
   entry WRITE(P : in   PRINTER; E : in ELEM);
   entry STOP_MULTIPLEXER;
private
   type PRINTER is new INTEGER range 1 .. 100;
end MULTIPLEXER;

task LINK_CONTROLLER(INTEGER range 1 .. 200) is
   entry SEND(P : in out PACKET);
   entry ACKNOWLEDGE;
end LINK_CONTROLLER;
```

```
task TRACK_MANAGER is
   type  TRACK is new INTEGER range 1..200;
   entry START(U : in USER; T : in TRACK);
   entry TRANSFER(TRACK'FIRST .. TRACK'LAST) (I : in ITEM);
end TRACK_MANAGER;

generic task KEYBOARD is
   entry READ (C : out  CHARACTER);
   entry WRITE(C : in   CHARACTER);
end KEYBOARD;

task MY_KEYBOARD is new KEYBOARD;
task USER;  -- a task with no visible part
```

Example of task declaration and body:

```
task PROTECTED_ARRAY is
   -- INDEX and ELEM are global types
   entry READ (I : in INDEX; V : out ELEM);
   entry WRITE(I : in INDEX; E : in  ELEM);
end;

task body PROTECTED_ARRAY is
   TABLE : array(INDEX'FIRST .. INDEX'LAST) of ELEM := (INDEX'FIRST .. INDEX'LAST => 0);
begin
   loop
      select
         accept READ (I : in INDEX; V  : out ELEM) do
            V := TABLE(I);
         end READ;
      or
         accept WRITE(I : in INDEX; E  : in  ELEM) do
            TABLE(I) := E;
         end WRITE;
      end select;
   end loop;
end PROTECTED_ARRAY;
```

9.2 Task Hierarchy

Tasks may be declared local to other task bodies, packages, subprograms, and blocks but not in the visible part of another task.

The elaboration of a task declaration creates one new potentially active thread of control (or, in the case of a family, one for each member of the family). This thread of control only becomes *active* when an initiate statement referring to the task is executed. Each thread of control has a parent, which is the thread of control which elaborated the corresponding task declaration (and is not necessarily the thread of control that executed the initiate statement). A thread of control can only exist if its parent thread of control is active.

Each elaboration of a task declaration can only create one thread (i.e. a task cannot be multiply active) and so the thread of control emanating from the elaboration of that task declaration may be referred to as the task without ambiguity. However, if a task is declared in a procedure, each call of that procedure creates a new thread of control. Normal scope rules prevent any ambiguity.

A procedure or entry in the visible part of a task can only be called if the task is active. If the task is not active, the TASKING_ERROR exception is raised in the task issuing the call.

Procedures and entries in the visible part of a family of tasks apply to each member of the family and are denoted by using the member name. Thus, outside the task body of F, the procedure P of the I-th member of the family F is denoted by

```
F(I).P
```

Types, constants, and exceptions apply to the family as a whole and are denoted by just using the family name. Thus, the type T declared in the visible part of the family F is denoted by

```
F.T
```

Within the task body, if selected components are used to denote items local to the task body, they must only mention the family name.

Notes:

- The main program is implicitly considered to be a task and therefore every thread of control is associated with a task.
- A subprogram can be used reentrantly by several threads of control.
- The implementation of task creation, although described in dynamic terms, can either be dynamic (storage for a task is allocated when the task is initiated) or static (storage for a task is allocated when the task declaration is elaborated). This is particularly relevant to task families which can be viewed in a similar manner to an access type; the index range gives an upper limit on the number of active tasks in the family. It is possible to influence the implementation of a given task declaration by providing an appropriate pragma in its declaration:

```
pragma CREATION(STATIC);
pragma CREATION(DYNAMIC);
```

9.3 Task Initiation

The execution of a task body is initiated by an initiate statement.

```
initiate_statement ::=
  initiate task_designator {, task_designator};

task_designator ::= task_name [(discrete_range)]
```

Execution of an initiate statement allows the execution of the designated tasks to begin in parallel with other currently active tasks. Tasks of a task family are individually initiated by following the family name by an appropriate index or collectively initiated by following the family name by a discrete range denoting all or part of the family.

On completion of an initiate statement the designated tasks are active. Initiation of a terminated task is possible and results in a new execution of the task. However, an attempt to initiate a task that is already active raises an INITIATE_ERROR exception in the task performing the initiate statement.

Examples:

```
initiate MULTIPLEXER, LINK_CONTROLLER(J), MY_KEYBOARD;
initiate LINK_CONTROLLER (3 .. N);
```

Notes:

If an initiate statement refers to more than one task, the tasks are made active simultaneously.

A task can (but need not) be initiated by its parent (the task elaborating its task declaration). For example, if several tasks are declared in the body of a parent task, one of them could be initiated by the parent task, or by another of the declared tasks, or by an outside task calling a visible procedure of the parent task. It is a consequence of the rules of the language that, in any case, a task cannot be initiated unless its parent task is active.

The parent of a task need not be the task lexically enclosing its declaration. For example, a task T could be declared in a procedure specified in the visible part of another task. The parent of the task T is then the task that calls the procedure and this, of course, need not be the enclosing task.

9.4 Normal Termination of Tasks

Normal termination of a task occurs when it reaches the end of its task body and when all locally declared tasks (if any) have terminated their execution. More generally, any subprogram, module, or block containing local task declarations cannot be left until all local tasks have terminated their execution.

9.5 Entry Declarations and Accept Statements

An entry declaration is similar to a subprogram declaration. For other tasks, the entry appears as a subprogram and it is called with the same syntax as subprogram calls. An entry declaration can also specify a family of identical entries, each denoted by an index from a discrete range. In this case every call must be subscripted by an index.

An entry can be declared either in a task specification or in the outermost declarative part of a task body, but not in any other declarative part; it is said to be *owned* by the corresponding task.

```
entry_declaration ::=
   entry identifier [(discrete_range)] [formal_part];

accept_statement ::=
   accept entry_name [formal part] [do
      sequence_of_statements
   end [identifier]];
```

An accept statement specifies the actions to be performed, if any, when the corresponding entry is called. There may be several accept statements corresponding to one entry.

Execution of an entry call, however, may be delayed until the task owning the entry reaches an accept statement for the corresponding entry. There are two possibilities:

(a) If a calling task issues an entry call before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is suspended.

(b) If a task reaches an accept statement prior to any call of that entry, the execution of the task is suspended until such a call occurs.

When an entry has been called and a corresponding accept statement is reached, the sequence of statements, if any, of the accept statement is executed by the called task in mutual exclusion. This interaction is called a *rendezvous*. Thereafter, the calling task and the task owning the entry can continue their execution in parallel.

If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued; there is only one queue associated with each entry. Each execution of an accept statement removes one call from the queue. The calls are processed in order of arrival. Each task can only be on one queue.

Entries may be overloaded both with each other and with procedures with the same identifier. An entry may be renamed as a procedure.

Restrictions on accept statements:

- A task can execute accept statements only for its own entries. Hence, an accept statement cannot appear in the statements of a procedure declared in the visible part of the task, nor can it appear in any procedure called directly or indirectly by such an externally visible procedure or by an internal task. An accept statement can, of course, occur in a subprogram called only by the task owning the entry.

- Initiation of a task may not be performed directly or indirectly by the sequence of statements of an accept statement.

Examples of entry declarations:

```
entry READ(V : out ELEM);
entry TRANSFER(TRACK'FIRST .. TRACK'LAST)(I : in ITEM);
```

Examples of entry calls:

```
READ(X);
KA.READ(Y);
LINK_CONTROLLER(3).ACKNOWLEDGE;
TRANSFER(TRACK_A)(J);
```

Examples of accept statements:

```
accept READ(V : out ELEM) do
   V := LOCAL_ELEM;
end READ;

accept TRANSFER(T)(I : in ITEM) do
   ...
end TRANSFER;
```

Notes:

An accept statement may contain other accept statements (possibly for the same entry) directly or indirectly. A task may call its own entries but it will, of course, deadlock. In contrast, a procedure declared in the visible part of a task can call local entries of the task, without risk of automatic deadlock when the procedure is called by other tasks.

9.6 Delay Statements

A delay statement suspends the task which executes it for at least the given time interval. This interval is expressed in the basic time unit of the clock. Time values may be expressed in terms of the predefined constant SECONDS, which gives the number of basic time units in one second. The type of the time interval is the predefined floating point type TIME.

```
delay_statement ::= delay simple_expression;
```

Example:

```
delay 3.0 * SECONDS;
```

A delay statement can occur wherever a statement is permitted.

9.7 Select Statement

A select statement allows a selective wait on one or more alternatives. The selection may depend on conditions associated with each alternative of the select statement.

```
select_statement ::=
    select
      [when condition =>]
        select_alternative
    { or [when condition =>]
        select_alternative}
    [ else
      sequence_of_statements]
    end select;

select_alternative ::=
      accept_statement  [sequence_of_statements]
    | delay_statement   [sequence_of_statements]
```

A select alternative is said to be open if there is no preceding when clause or if the corresponding condition is true. It is said to be closed otherwise.

Execution of a select statement proceeds as follows:

(a) All conditions are first evaluated in textual order to determine which alternatives are open.

(b) An open alternative starting with an accept statement may be selected if a corresponding rendezvous is possible (i.e. when a corresponding entry call has been issued by another task). When such an alternative is selected, the corresponding accept statement and possible subsequent statements are executed.

(c) An open alternative starting with a delay statement will be selected if no other alternative has been selected before the specified time interval has elapsed. Any subsequent statements of the alternative are then executed.

(d) If no alternative can be immediately selected, and there is an else part, the else part is executed. If there is no else part, the task waits until an open alternative can be selected.

(e) If all alternatives are closed and there is an else part, the else part is executed. If there is no else part, the exception SELECT_ERROR is raised.

In general, several entries of a task may have been called before a select statement is encountered. As a result, several alternative rendezvous are possible. Similarly, several open alternatives may start with an accept statement for the same entry. In such cases one of these alternatives is selected at random. If several open alternatives start with a delay statement, only the one with the shortest time interval is considered.

A select statement cannot contain both an else part and alternatives starting with delay statements. A select statement must contain at least one alternative commencing with an accept statement and so its position is consequently constrained in the same manner as the accept statement (see 9.5).

Example:

```
task READER_WRITER is
   procedure READ (V : out ELEM);
   entry WRITE(E : in ELEM);
end;

task body READER_WRITER is
   RESOURCE  : ELEM;
   READERS   : INTEGER := 0;

   entry START;
   entry STOP;

   procedure READ(V : out ELEM) is
      -- READ is a procedure, not an entry, hence concurrent calls of READ are possible
      -- READ synchronizes such calls with the entry calls START and STOP
   begin
      START; V := RESOURCE; STOP;
   end;

begin
   accept WRITE(E : in ELEM) do
      RESOURCE := E;
   end;

   loop
      select
         accept START;
         READERS := READERS + 1;
      or
         accept STOP;
         READERS := READERS - 1;
      or when READERS = 0 =>
            accept WRITE(E : in ELEM) do
               RESOURCE := E;
            end WRITE;
      end select;
   end loop;
end READER_WRITER;
```

9.8 Task Priorities

Each task has an associated priority, which is an integer value of the implementation defined subtype PRIORITY, defined as:

```
subtype PRIORITY is INTEGER range
   SYSTEM'MIN_PRIORITY .. SYSTEM'MAX_PRIORITY;
```

A lower value indicates a lower degree of urgency. The main program of a system is started with an implementation defined intermediate priority. Whenever a task is initiated, it takes the priority of its initiator at that time. A task can set its own priority to some value P by a call of the predefined procedure SET_PRIORITY thus:

```
SET_PRIORITY(P);
```

Notes:

There may be several tasks that are ready to be executed by the system processors. In choosing the processes to be executed, processes with the highest priority are treated first. Processes of the same priority level are treated on a first in, first out basis. The language does not specify when a scheduling decision is made. For example, a round-robin time sliced strategy is acceptable.

Priorities should only be used to indicate degrees of urgency. They should not be used for task synchronization.

9.9 Task and Entry Attributes

A task T has the following predefined attributes:

```
T'ACTIVE    -- equal to TRUE if the task is active, FALSE otherwise
T'PRIORITY  -- the current priority of the task T
T'CLOCK     -- the cumulative processing time of the task T
```

The real time system clock can be accessed with the attribute SYSTEM'CLOCK. The cumulative processing time of a task is initialized to zero when the task is initiated. The attributes T'CLOCK and SYSTEM'CLOCK are of type TIME.

When a task of a family F needs to reference its own index, for example to pass it to another task, it may use the attribute F'INDEX for that purpose. This attribute cannot be used in the visible part of the family.

For an entry E, the attribute E'COUNT gives the number of calls to the entry that have not yet been serviced. This attribute can only be used in the body of the task owning the entry.

9.10 Abort Statements

Abnormal termination of a task is caused by an abort statement.

```
abort_statement ::= abort task_designator {,task_designator};
```

An abort statement causes the unconditional asynchronous termination of the tasks mentioned in the list of task designators. If a task is not active (i.e. not yet initiated or already terminated), there is no effect. Abnormal termination of a task causes the abnormal termination of all tasks of which it is the direct or indirect parent.

On completion of the abort statement the designated tasks are no longer active. If a designated task is waiting on an entry queue then it is merely removed from the queue. However, if it is already engaged in a rendezvous, the other task receives a TASKING_ERROR exception.

Example:

```
abort TRACK_MANAGER, LINK_CONTROLLER(1 .. 10);
```

Notes:

An abort statement should only be used in extremely severe situations requiring unconditional termination. In less extreme cases (where the task to be terminated can be given the possibility of executing some cleanup actions before termination), the exception FAILURE could be raised for the task (see 11.5). A task may abort any task including itself and its parent.

9.11 Signals and Semaphores

Two generic tasks, named SIGNAL and SEMAPHORE, are predefined in the language. Their semantics correspond to the following declarations:

```
generic task SIGNAL is
   entry SEND;
   entry WAIT;
end SIGNAL;

task body SIGNAL is
   RECEIVED : BOOLEAN := FALSE;
begin
   loop
      select
         accept SEND;
         RECEIVED := TRUE;
      or when RECEIVED =>
         accept WAIT;
         RECEIVED := FALSE;
      end select;
   end loop;
end SIGNAL;
```

```
generic task SEMAPHORE is
   entry P;
   entry V;
end SEMAPHORE;

task body SEMAPHORE is
begin
   loop
      accept P;
      accept V;
   end loop;
end SEMAPHORE;
```

Example of use of a semaphore:

```
task SEMA is new SEMAPHORE;
...
initiate SEMA;
...
SEMA.P;
   -- mutual exclusion
SEMA.V;
```

Although the task declarations above are given in the language, for the sake of semantic description, their being predefined authorizes an implementation to recognize them and implement them by making an optimal use of the facilities provided by the machine or the underlying system.

9.12 Example of Tasking

The following example defines a buffering task to smooth variations between the speed of output of a producing task and the speed of input of some consuming task. For instance, the producing task may contain the statements

```
loop
   -- produce the next character CHAR
   BUFFER.WRITE(CHAR);
   exit when CHAR = END_OF_TRANSMISSION;
end loop;
```

and the consuming task may contain the statements

```
loop
   BUFFER.READ(CHAR);
   -- consume the character CHAR
   exit when CHAR = END_OF_TRANSMISSION;
end loop;
```

The buffering task contains an internal pool of characters processed in a round-robin fashion. The pool has two indices, an IN_INDEX denoting the space for the next input character and an OUT_INDEX denoting the space for the next output character.

```
task BUFFER is
   entry READ (C : out  CHARACTER);
   entry WRITE(C : in   CHARACTER);
end;

task body BUFFER is
   POOL_SIZE  : constant INTEGER := 100;
   POOL       : array(1 .. POOL_SIZE) of CHARACTER;
   COUNT      : INTEGER range 0 .. POOL_SIZE := 0 ;
   IN_INDEX, OUT_INDEX : INTEGER range 1 .. POOL_SIZE := 1;
begin
   loop
      select
         when COUNT < POOL_SIZE =>
            accept WRITE(C : in CHARACTER) do
               POOL(IN_INDEX) := C;
            end;
            IN_INDEX  := IN_INDEX mod POOL_SIZE + 1;
            COUNT     := COUNT + 1;
      or when COUNT > 0 =>
            accept READ(C : out CHARACTER) do
               C := POOL(OUT_INDEX);
            end;
            OUT_INDEX  := OUT_INDEX mod POOL_SIZE + 1;
            COUNT      := COUNT - 1;
      end select;
   end loop;
end BUFFER;
```

10. Program Structure and Compilation Issues

This chapter describes the overall structure of programs and the facilities for separate compilation of their parts. A program is a collection of one or more compilation units. These can be subprogram bodies, module specifications, or module bodies. The body of a unit declared within another unit can be separately compiled as a subunit.

10.1 Compilation Units

A program can be compiled as a single compilation unit or it can be submitted to the compiler as a succession of compilation units. One compilation can consist of several such units. The compilation units of a program are said to belong to a *program library*.

compilation ::= {compilation_unit}

compilation_unit ::=
  [visibility_restriction][**separate**] unit_body

Each compilation unit is in effect a restricted program unit. In the absence of an explicit visibility restriction, an empty visibility list is assumed. The visibility rules that apply to compilation units follow from those that apply to all restricted program units (see 8.3). In particular, a separately compiled unit that makes use of a separately compiled module must name that module in its visibility list. These dependencies between units have an influence on the order of compilation and recompilation of compilation units.

All compilation units (that are not subunits) belonging to the same program library must have different names.

A compilation unit that is a subprogram body can be a main program in the usual sense. The means by which main programs are executed are not within the language definition.

Example 1:

A compilation unit can be split into a number of compilation units. For example, consider the following program.

Example 1a: *Single compilation unit*:

```
procedure PROCESSOR is

   package D is
      LIMIT  : constant INTEGER := 1000;
      TABLE  : array (1 .. LIMIT) of INTEGER;
      procedure RESTART;
   end D;

   package body D is
      procedure RESTART is
      begin
         for I in 1 .. LIMIT loop
            TABLE(I) := I;
         end loop;
      end;
   begin
      RESTART;
   end D;

   procedure Q(X : INTEGER) is
      use D;
   begin
      ...
      TABLE(X) := TABLE(X) + 1;
      ...
   end Q;

begin
   ...
   D.RESTART;   -- reinitializes TABLE
   ...
end PROCESSOR;
```

The following three compilation units define a program with an equivalent effect (the broken lines between compilation units are here to remind the reader that these units need not be contiguous texts).

Example 1b: *Several compilation units*:

```
package D is
   LIMIT  : constant INTEGER := 1000;
   TABLE  : array (1 .. LIMIT) of INTEGER;
   procedure RESTART;
end D;
```

```
-----------------------------------------------

package body D is
   procedure RESTART is
   begin
      for I in 1 .. LIMIT loop
         TABLE(I) := I;
      end loop;
   end;
begin
   RESTART;
end D;

-----------------------------------------------

restricted(D)
procedure PROCESSOR is
   procedure Q(X : INTEGER) is
      use D;
   begin
      ...
      TABLE(X) := TABLE(X) + 1;
      ...
   end Q;
begin
   ...
   D.RESTART;   -- reinitializes TABLE
   ...
end PROCESSOR;
```

Note that in the latter version, the package D is (implicitly) a fully restricted program unit. Hence, it has no visibility of outer identifiers other than the predefined identifiers. In particular, D does not depend on any identifier declared in PROCESSOR and hence can be extracted from PROCESSOR.

The procedure PROCESSOR is also a restricted unit, but must name D in its visibility list in order to contain a legal use clause for D.

These three compilation units can be submitted in one or more compilations. For example, it is possible to submit the package specification and the package body in a single compilation.

Example 2: A complete program

The following is an example of a complete program to print the real roots of a quadratic equation. The packages MATH_LIB and TEXT_IO (as the package D in the previous example) may be used by different main programs. These packages are assumed to be already present in the program library.

```
restricted(MATH_LIB, TEXT_IO)
procedure QUADRATIC_EQUATION is
   use TEXT_IO;
   A, B, C, D : FLOAT;
begin
   GET(A); GET(B); GET(C);
   D := B**2 - 4.0*A*C;
   if D < 0.0 then
      PUT("IMAGINARY ROOTS");
   else
      declare
         use MATH_LIB;  -- note: SQRT is defined in MATH_LIB
      begin
         PUT("REAL ROOTS :  ");
         PUT((B - SQRT(D))/(2.0*A));
         PUT((B + SQRT(D))/(2.0*A));
         PUT(NEWLINE);
      end;
   end if;
end QUADRATIC_EQUATION;
```

Notes:

A module that is a compilation unit (such as D or MATH_LIB) can depend on other separately compiled modules.

A visibility restriction need only mention the modules that are actually used within a compilation unit. It need not (and should not) mention other modules on which the modules of the visibility list depend, unless these other modules are directly used in the compilation unit. For example, the implementation of the package INPUT_OUTPUT may need the operations provided by a more basic package. The latter should not appear in the visibility list of QUADRATIC_EQUATION, since these operations are not directly called within its body.

A compilation unit can be a generic program unit.

10.2 Subunits of Compilation Units

The body of a subprogram or module declared in the outermost declarative part of another compilation unit (or subunit) can be separately compiled and is then said to be a subunit. Within the subprogram or module where a subunit is declared, its body is represented by a body stub at the place where the body would otherwise appear. This method of splitting a program permits hierarchical program development.

```
body_stub ::=
      subprogram_specification is separate;
    | module_nature body identifier is separate;
```

A subunit is said to be *enclosed* by the compilation unit where its stub is given. Transitively, a subunit of a subunit of a unit is also said to be enclosed by the unit.

The body of a subunit must have a visibility restriction, itself followed by the reserved word **separate** (see 10.1). The first name appearing in the visibility list must be the name of an *enclosing* compilation unit. The name of a subunit is local to its immediately enclosing unit. In consequence, several subunits of the same name can exist within a program library.

Example 3a:

The procedure TOP is first written as a compilation unit without subunits.

```
procedure TOP is
  type REAL is digits 10;
  R, S : REAL;

  package D is
    PI : constant REAL:= 3.14159_26536;
    function   F (X: REAL) return REAL;
    procedure  G (Y, Z: REAL);
  end D;

  package body D is
    -- some local declarations of D followed by
    function F(X : REAL) return REAL is
    begin
      -- sequence of statements of F
    end F;

    restricted(TOP, INPUT_OUTPUT)
    procedure G(Y, Z : REAL) is
    begin
      -- sequence of statements of G
    end G;
  end D;


  procedure Q(U : in out REAL) is
    use D;
  begin
    ...
    U := F(U);
    ...
  end Q;
begin  -- TOP
  ...
  Q(R);
  ...
  D.G(R, S);
  ...
end TOP;
```

Example 3b:

The package body D and the body of the subprogram Q can be made into separate subunits of TOP as follows:

```
procedure TOP is

  type REAL is digits 10;
  R, S : REAL;

  package D is
    PI : constant REAL := 3.14159_26536;
    function   F (X : REAL) return REAL;
    procedure  G (Y, Z : REAL);
  end D;

  package body D is separate;                    -- stub of D
  procedure Q(U : in out REAL) is separate;      -- stub of Q
begin  -- TOP
  ...
  Q(R);
  ...
  D.G(R, S);
  ...
end TOP;

-------------------------------------------------

restricted(TOP)
separate procedure Q(U : in out REAL) is
  use D;
begin
  ...
  U := F(U);
  ...
end Q;

-------------------------------------------------

restricted (TOP)
separate package body D is
  -- some local declarations of D followed by

  function F(X : REAL) return REAL is
  begin
    -- sequence of statements of F
  end F;

  procedure G(Y, Z : REAL) is separate;    -- stub of G
end D;
```

```
-----------------------------------------------

restricted(TOP, INPUT_OUTPUT)
separate procedure G(Y, Z : REAL) is
begin
   -- sequence of statements of G
end G;
```

In the above example, Q and D are subunits of TOP (TOP encloses Q and D); G is a subunit of D (D encloses G and similarly TOP encloses G). The visibility list of G must mention TOP since G accesses the type REAL (mentioning D instead of TOP would not be enough).

Note that the visibility lists in the split version are established in such a manner that the same identifiers are visible at all program points as in the initial version. For example, the variables R and S declared in TOP, the constant PI declared in the visible part of D and the other entities declared in the package body D are all visible within the sequence of statements of the subunit G.

10.3 Order of Compilation

The visibility rules that apply to compilation units (whether subunits or not) are the usual rules that apply to all restricted program units.

The rules defining the order in which units can be compiled are direct consequences of the visibility rules. A unit must be compiled after all compilation units whose names appear in its visibility list or in the visibility list of any textually nested subprogram or module. A module body must be compiled after the corresponding module specification. The subunits of a unit must be compiled after the unit.

Consistent with the partial ordering defined above, the compilation units of a program can be compiled in any order.

In the previous examples:

(a) The package body D must be compiled after the corresponding package specification (example 1b).

(b) The specification of the package D must be compiled before the procedure PROCESSOR. On the other hand, the procedure PROCESSOR can be compiled either before or after the package body D.

(c) The procedure QUADRATIC_EQUATION (example 2) must be compiled after the library modules MATH_LIB and INPUT_OUTPUT that appear in its visibility list. Similarly (example 3a) the procedure TOP must be compiled after the library module INPUT_OUTPUT that appears in the visibility list of the nested procedure G. On the other hand, in example 3b INPUT_OUTPUT could be compiled after TOP.

(d) The subunits Q and D (example 3b) must be compiled after the compilation unit TOP. Similarly the subunit G must be compiled after the enclosing unit D. Note also that the library module INPUT_OUTPUT must be compiled before G.

Similar rules apply for recompilations. Any change in a given compilation unit can only affect its subunits and other compilation units mentioning the unit in their visibility lists. Hence the potentially affected units need to be recompiled. An implementation may be able to reduce the recompilation costs if it can deduce that some of the potentially affected units are not actually affected by the change.

Note that the subunits of a unit can always be recompiled without affecting the unit itself. Similarly, changes in a module body do not affect other (non-nested) units, since these units only have access to the visible part of the module. Hence to minimize recompilations, it is advantageous to compile the module body and the module specification (the visible part) in different compilations.

10.4 Program Library

Compilers must preserve the same degree of type safety for a program consisting of several compilation units and subunits, as for a program submitted as a single compilation unit. Consequently a library file containing information on the compilation units of the program library must be maintained by the compiler. This information may include symbol tables and other information pertaining to the order of previous compilations.

A normal submission to the compiler consists of the compilation unit(s) and the library file. The latter is used for checks and is updated as a consequence of the current compilation.

There should be compiler commands for creating the program library of a given program or of a given family of programs. These commands may permit the reuse of units of other program libraries. Finally, there should be commands for interrogating the status of the units of a program library. The form of these commands is not specified by the language definition.

10.5 Elaboration of Compilation Units

Before the execution of a main program, all library modules that are not subunits and that are used by the main program are elaborated. These modules are units mentioned in the visibility lists of the main program and of its subunits, and transitively in the visibility lists of these library modules themselves.

The elaboration of these modules is performed consistently with the partial ordering defined by the visibility lists (see 10.3).

10.6 Program Optimization

A static expression can be evaluated by the compiler. In consequence, if a static expression is required and the actual expression involves a variable, or if an exception arises in the evaluation of the expression, then the program is in error. On the other hand, a compiler may be able to optimize a program by evaluating expressions which are not required to be static. If the evaluation raises an exception, then the code in that path in the program can be replaced by code to raise the exception. Under such circumstances, the compiler may warn the programmer of a potential error.

Optimization of the elaboration of declarations and the execution of statements may be performed by compilers. If a subprogram is compiled by an in-line substitution of the body, then expressions within the body may be capable of further optimization as above.

A compiler may find that some statements or subprograms cannot be executed, in which case the corresponding code can be omitted. If non-static expressions within such code would generate an exception, then the program is not in error. These rules permit the effect of *conditional compilation* within the language.

11. Exceptions

This chapter defines the facilities for dealing with errors or other exceptional situations that arise during program execution. An *exception* is an event that causes suspension of normal program execution. Bringing an exception to attention is called *raising* the exception. To execute some actions, in response to the occurrence of an exception, is called *handling* the exception.

The units whose execution can be prematurely terminated by an exception are blocks, subprograms, and modules. Exceptions are introduced by exception declarations. Exceptions can be raised explicitly by raise statements, or they can be propagated by subprograms, blocks, or language defined operations that raise the exceptions. When an exception occurs, control can be passed to a user-provided exception handler.

11.1 Exception Declarations

An exception declaration defines one or several exceptions whose names can appear in raise statements and in exception handlers within the scope of the declaration.

exception_declaration ::= identifier_list : **exception**;

The identity of the exception introduced by an exception declaration is established at compilation time (exceptions can be viewed as constants of some predefined enumeration type initialized with static expressions). Hence an exception declaration introduces only one exception even if it is declared in a recursive procedure.

Examples of user-defined exception declarations:

```
SINGULAR     : exception;
END_OF_FILE  : exception;

STACK_OVERFLOW, STACK_UNDERFLOW : exception;
```

The following exceptions are predefined in the language:

ACCESS_ERROR	When an access variable has the value **null** and an attempt is made to read or to update the designated dynamic object (see 3.8).
ASSERT_ERROR	When violating an assertion (see 5.9).
DISCRIMINANT_ERROR	When attempting to access a component of a variant part not prescribed by the record's discriminant (see 4.1.2).
DIVIDE_ERROR	When dividing a number by zero (see 4.5.5, 4.5.6).
FAILURE	For general use within procedures and tasks. This is the only exception that can be raised by a task for another task (see 11.3, 11.5).
INDEX_ERROR	When an index value is outside the range specified for the array (see 4.1.1).
INITIATE_ERROR	When attempting to initiate a task that is already active (see 9.3).
NO_VALUE_ERROR	When accessing the value of an uninitialized variable or returning from a function without a value (see 6.5).
OVERFLOW	When an arithmetic operation fails by attempting to produce a value which is too large to be handled by the implementation (see 4.5).
OVERLAP_ERROR	When attempting to assign overlapping slices (see 5.1.1).
RANGE_ERROR	When exceeding the declared range of a variable or type (see 4.5).
SELECT_ERROR	When all alternatives of a select statement without else part are closed (see 9.7).
STORAGE_OVERFLOW	When the dynamic storage allocated to a task is exceeded, or during the execution of an allocator, if the available space for the collection of dynamic objects is exhausted (see 13.2).
TASKING_ERROR	When exceptions arise during intertask communication (see 9.2, 11.4).
UNDERFLOW	When a floating point operation fails by attempting to produce a value which is too small to be handled by the implementation (see 4.5.5).

11.2 Exception Handlers

The processing of one or more exceptions is specified by an exception handler. A handler may appear at the end of a unit which must be a block, subprogram body, or module body. The word *unit* will have this meaning in this section.

```
exception_handler ::=
  when exception_choice {| exception_choice} =>
    sequence_of_statements

exception_choice ::= exception_name | others
```

Each handler handles the named exceptions when they are raised in the given unit. An alternative containing the choice **others** applies to all exceptions not listed in other alternatives, including exceptions whose names are not visible within the current unit.

When an exception is raised within a unit, either during elaboration of its local declarations, or during the execution of its sequence of statements, the execution of the corresponding handler replaces the execution of the remainder of the unit: the actions following the point where the exception is raised are skipped, and the execution of the handler terminates the execution of the unit. If no handler is provided for the exception, the unit is terminated and the exception is propagated according to the rules stated in section 11.3.1.

Since a handler acts as a substitute for the corresponding unit, the handler has, in general, the same capabilities as the unit it replaces. For example, a handler within a function has access to its parameters and may issue a return statement on behalf of the function. However, since an exception may be raised during the elaboration of the declarations local to the unit considered, it cannot be assumed within a handler that all declarations have been elaborated.

Example:

```
begin
   -- sequence of statements
exception
   when SINGULAR | OVERFLOW =>
      PUT(" MATRIX IS SINGULAR ");
   when others =>
      PUT(" FATAL ERROR ");
      raise FAILURE;
end;
```

11.3 Raise Statements

An exception can be explicitly raised by a raise statement.

raise_statement ::= **raise** [*exception*_name];

A raise statement raises the named exception. A task can raise the predefined exception FAILURE in another task (say T) by giving T.FAILURE as exception name. A raise statement of the form

raise;

can only appear in a handler. It reraises the same exception which caused transfer to the handler.

Examples:

```
raise;
raise SINGULAR;
raise MULTIPLEXER.FAILURE;
raise LINK_CONTROLLER(5).FAILURE;
raise OVERFLOW;   -- explicitly raising a predefined exception
```

11.3.1 Dynamic Association of Handlers with Exceptions

When an exception is raised, normal program execution is suspended and one of the following events takes place.

(a) If a block does not contain a local handler for the exception, execution of the block is terminated and the same exception is reraised in the enclosing sequence of statements. Similarly, if a subprogram does not contain a local handler, its execution is terminated and the exception is reraised at the point of call of the subprogram. In both cases the exception is said to be *propagated*. The predefined exceptions are exceptions that can be propagated by the language defined constructs.

(b) If a task does not contain a local handler for the exception the task is terminated but the exception is not propagated.

(c) If a local handler has been provided, execution of the handler replaces execution of the remainder of the current unit. A further exception raised in the sequence of statements of the handler causes termination of the current unit, and the exception is propagated if the current unit is a block or subprogram as in case (a).

Example:

```
procedure P is
   ERROR : exception;
   procedure R;

   procedure Q is
   begin
      R;
      ...                        -- exception possibility(2)
   exception
      ...
      when ERROR =>              -- handler E2
      ...
   end Q;

   procedure R is
   begin
   ...                           -- exception possibility(3)
   end R;

begin
   ...                           -- exception possibility(1)
   Q;
   ...
exception
   ...
   when ERROR =>                 -- handler E1
   ...
end P;
```

The following cases can arise:

(1) If the exception ERROR is raised in the statement list of the outer procedure P, the handler E1 provided within P is used to complete the execution of P.

(2) If the exception ERROR is raised in the statement list of Q, the handler E2 provided within Q is used to complete the execution of Q. Control will be returned to the point of call of Q upon completion of the handler.

(3) If the exception ERROR is raised in the body of R, called by Q, the execution of R is terminated and the same exception is raised in the body of Q. The handler E2 is then used to complete the execution of Q, as in case (2).

The third case results in a dynamic binding, since the exception raised in R results in passing control to a local handler in Q that is not visible from R. Note also that if a handler were provided within R for the choice **others**, case 3 would cause execution of this alternative, rather than direct termination of R.

Lastly, if ERROR had been declared inside R, rather than in P, the handlers E1 and E2 could not provide an explicit handler for ERROR since this identifier would not be visible within the bodies of P and Q. In case 3, the exception could however be handled in Q by providing a handler for the choice **others**.

12. Generic Program Units

Subprograms and modules can be generic. Generic programs units may be thought of as (possibly parameterized) models of program units; as such they cannot be used directly. For example, a generic subprogram cannot be called. Instances (i.e. copies) of the model are obtained by generic instantiation. These are ordinary subprograms and modules that can be used directly.

A subprogram or module can be designated as generic by the inclusion of a generic clause in its specification. A generic clause can include the definition of generic parameters. An instance of a generic unit with appropriate actual parameters for the generic formal parameters, is obtained as the result of a subprogram or module declaration with a generic instantiation.

12.1 Generic Clauses

A generic clause given with a subprogram or module specification specifies that the unit is generic and defines any generic parameters.

```
generic_clause ::=
   generic [(generic_parameter {; generic_parameter})]

generic_parameter ::=
      parameter_declaration
   |  subprogram_specification [is [name.]designator]
   |  [restricted] type identifier
```

The usual forms of parameter declarations available for subprogram specifications can also appear in generic clauses.

Within an instantiated unit, an *in* or *in out* parameter provides access to the value of the actual parameter, an *out* or *in out* parameter permits assignment to the variable given as actual parameter, as usual. In addition, generic parameters can denote types and subprograms.

A type given as a generic parameter is considered as a private type within the body of the generic unit. Hence, if any operation (apart from assignment and comparison for equality or inequality) on objects of this type is to be used in the generic body, the operation must also be provided as an additional generic parameter. Neither assignment nor the predefined comparison for equality or inequality is available if the generic type parameter is specified as restricted.

On the other hand, a generic parameter of a generic clause cannot refer to a previous generic parameter of the same clause unless this previous generic parameter denotes a type. Both the specification and the body of a generic subprogram or module can refer to generic parameters.

Expressions appearing in a generic clause are evaluated during the elaboration of the clause unless they refer to a type that is a generic parameter (for example, an expression that is an attribute of a type); such expressions are evaluated during elaboration of generic instantiations.

The specification of a generic parameter that is a subprogram may provide a designator to be used by default upon instantiation. An actual parameter is optional in this case; in the absence of an actual parameter the designator supplied in the generic clause is used. Such a default designator can be any subprogram matching the corresponding subprogram specification; it can be an attribute of a type that is a generic parameter. Note that the designator can be prefixed by the name of a type of which it is an attribute.

A non-local name in the body of a generic unit is identified during the elaboration of the generic body. A generic unit can be separately compiled.

Examples of generic clauses:

```
generic -- parameterless
generic(SIZE : INTEGER;  type ELEM)
generic(LENGTH : INTEGER := 200) -- default value

generic(type T;
        function "*"(X,Y: T) return T)
generic(type T;
        function "*"(X,Y: T) return T is T."*")    -- default operator
```

Examples of generic subprograms:

```
generic(type ELEM)
procedure EXCHANGE(U, V: in out ELEM) is
   T : ELEM;
begin
   T := U; U := V; V := T;
end EXCHANGE;

generic(type T;
        function "*"(U, V: T) return T is T."*")
function SQUARING(X: T) return T is
   pragma INLINE;
begin
   return X * X;
end SQUARING;
```

Example of generic module:

```
generic task SEMAPHORE is
   entry P;
   entry V;
end;

task body SEMAPHORE is
begin
   loop
      accept P;
      accept V;
   end loop;
end;
```

Notes:

An explicit formal parameter for the equality operation may be supplied. In the case of an unrestricted type it will override the built-in comparison operation. In the case of a restricted type, it allows and provides the comparison operation.

12.2 Generic Instantiation

An instance of a generic program unit is obtained as the result of the elaboration of a subprogram or module declaration defined in terms of a generic instantiation.

```
generic_instantiation ::=
  new name [(generic_association {, generic_association})]

generic_association ::=
    parameter_association
  | [formal_parameter is] [name.]designator
  | [formal_parameter is] type_mark
```

The forms of declarations with generic instantiations are given in the sections on subprogram declarations (6.2) and modules (7.1):

```
subprogram_nature designator is generic_instantiation;
module_nature identifier [(discrete_range)] is generic_instantiation;
```

Actual parameters must be supplied for each generic formal parameter unless the corresponding generic clause specifies a default. Parameters can be given in positional form or in named form as for subprogram calls (see 5.2). Each actual parameter must match the corresponding generic formal parameter. A type matches a type; a restricted actual type does not match an unrestricted formal type. For subprogram parameters all occurrences of the name of a type that is a formal generic parameter are replaced by the corresponding actual parameter. An actual subprogram matches a formal subprogram having parameters with the same order, mode, type, and constraint and with the same result type and constraint (the parameter names and default values being ignored).

An actual parameter of a generic association is evaluated during elaboration of the generic instantiation. If a formal generic parameter is used in the generic body in a context which requires static evaluation, the corresponding actual parameter must be a static expression. In particular, this rule applies to default expressions given for optional **in** parameters.

The elaboration of a generic instantiation creates an instance of the generic unit in which all generic parameters are replaced as defined above by the parameters supplied in the generic associations. The specification of a module or subprogram obtained by generic instantiation is derived from the specification of the corresponding generic unit after the parameter replacements.

Recursive instantiation of generic units is not allowed.

Examples of declarations with generic instantiations:

```
procedure SWAP is new EXCHANGE(ELEM is INTEGER);
procedure SWAP is new EXCHANGE(CHARACTER);        -- SWAP is overloaded
```

```
function SQUARE is new SQUARING(INTEGER);          -- "*" of INTEGER used by default
function SQUARE is new SQUARING(MATRIX,MATRIX_PRODUCT);

task SEMA is new SEMAPHORE;
```

Examples of the use of instantiated units:

```
SWAP(X, Y);
I := SQUARE(8);
SEMA.P;
```

12.3 Example of a Generic Package

The following example provides a possible formulation of stacks formulated as a generic package. The size of each stack and the type of the stack elements are provided as generic parameters.

```
generic(SIZE: INTEGER; type ELEM)
package STACK is
   procedure PUSH (E: in    ELEM);
   procedure POP  (E: out   ELEM);
   OVERFLOW, UNDERFLOW : exception;
end STACK;

package body STACK is

   SPACE   : array (1 .. SIZE) of ELEM;
   INDEX   : INTEGER range 0 .. SIZE := 0;

   procedure PUSH(E: in ELEM) is
   begin
      if INDEX = SIZE then
         raise OVERFLOW;
      end if;
      INDEX := INDEX + 1;
      SPACE(INDEX) := E;
   end PUSH;

   procedure POP(E: out ELEM) is
   begin
      if INDEX = 0 then
         raise UNDERFLOW;
      end if;
      E := SPACE(INDEX);
      INDEX := INDEX - 1;
   end POP;

end STACK;
```

Instances of this generic module can be obtained as follows:

```
package STACK_INT    is new STACK(SIZE := 200, ELEM is INTEGER);
package STACK_BOOL   is new STACK(100, BOOLEAN);
```

Thereafter, the procedures of the instantiated packages can be called as follows:

```
STACK_INT.PUSH(I);
STACK_BOOL.PUSH(TRUE);
```

Alternatively, a generic formulation of the type STACK can be formulated as follows (package body omitted):

```
generic(SIZE : INTEGER; type ELEM)
package ON_STACKS is
  type STACK is private;
  procedure PUSH (S: in out STACK; E: in     ELEM);
  procedure POP  (S: in out STACK; E: out    ELEM);
  OVERFLOW, UNDERFLOW : exception;
private
  type STACK is
    record
      SPACE : array(1 .. SIZE) of ELEM;
      INDEX : INTEGER range 0 .. SIZE := 0;
    end record;
end;
```

In order to use such a package, an instantiation must be created and thereafter stacks of the corresponding type can be declared as:

```
package STACK_INT is new ON_STACKS(SIZE := 100, ELEM is INTEGER);
```

An example of the use of the instantiated package is as follows:

```
declare
  use STACK_INT;
  S : STACK;
begin
  ...
  PUSH(S, 20);
  ...
end;
```

13. Representation Specifications and Implementation Dependent Features

Representation specifications specify the mapping between data types and features of the underlying machine that executes programs. Representation specifications can be more or less direct: in some cases, they completely specify the mapping, in other cases they only provide criteria for choosing a mapping.

Mappings acceptable to an implementation do not alter the net effect of a program. They can be provided to give a more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware).

```
representation_specification ::=
    packing_specification          | length_specification
  | record_type_representation     | enumeration_type_representation
  | address_specification
```

Representation specifications must appear immediately after the list of declarations of a declarative part, and can only apply to items declared in the same declarative part. A representation specification given for a type applies to all objects of the type. In the absence of explicit specifications, representations are determined by the compiler.

All representation specifications must be determinable at compilation time. In particular, expressions appearing in such specifications must be static expressions. Depending on the specification, such expressions represent either a number of bits or a number of storage units.

For record and enumeration types derived from other similar types, a representation specification is legal only if the derived type does not derive user defined subprograms from its parent type (see 3.4). In addition, implementations may limit representation specifications to those that can be simply handled by the underlying hardware.

13.1 Packing Specifications

A packing specification indicates that storage minimization should be the main criterion for selecting the representation of a record or array type.

```
packing_specification ::= for type_name use packing;
```

This means that gaps between the storage places allocated to consecutive components should be minimized. However, it does not affect the mapping of each component on storage. This mapping can only be influenced by a representation specification for the component type.

The predefined type STRING is packed.

Examples:

```
for MATRIX use packing;
for FILE_DESCRIPTOR use packing;
```

13.2 Length Specifications

A length specification controls the amount of storage associated with a named entity.

length_specification ::= **for** name **use** *static*_expression;

The name must be one of the following:

(a) Name of a type that is not an access type:

The value of the expression specifies the maximum number of bits to be allocated to objects of the type. This number must be at least equal to the minimum needed for the representation of objects of the type. This form of length specification can be used to achieve a biased representation.

(b) Name of an access type:

The value of the expression is the number of bits to be reserved for the collection, i.e. the space for all objects of that access type.

Note that dynamic objects allocated in a collection need not occupy the same storage if they are records with variants or dynamic arrays. Note also that the allocator itself may require some space. Hence, the length specification does not always give precise control over the maximum number of allocated objects.

(c) Name of a task:

The value of the expression is the number of bits to be reserved for an activation of the task. The method of allocation is not defined (for example, a stack, a general storage allocator, or fixed storage could be used).

The exception STORAGE_OVERFLOW is raised if a task or access type exceeds the reserved space.

Examples:

```
-- assumed declarations:

type BIASED is new INTEGER range 10_000 .. 10_255;
type SHORT is delta 0.01 range -100.0 .. 100.0;
BYTE : constant INTEGER := 8;
PAGE : constant INTEGER := 1000 * SYSTEM'STORAGE_UNIT;
```

```
-- length specifications:

for COLOR     use 1*BYTE;
for ELEMENT   use INTEGER'SIZE;
for BIASED    use 1*BYTE;          -- biased representation

for PRINTER   use 4*PAGE;

for CAR       use 2000*CAR'SIZE;   -- space for approximately 2000 cars

for SHORT     use 15;
```

Notes:

In the last example, 15 bits is actually the minimum number of necessary bits for SHORT, since a sign, 7 bits above the point, and 7 below the point are needed. An implementation need not provide the ability to reserve just the minimum, because of the masking code needed.

13.3 Enumeration Type Representations

An enumeration type representation specifies the internal codes for the literals of an enumeration type.

enumeration_type_representation ::= **for** *type*_name **use** aggregate;

The aggregate used to specify this mapping is an array aggregate of type

array (*type*_name) **of** INTEGER

All enumeration literals must be provided with distinct integer codes, and the aggregate must be a static expression. The integer codes specified for the enumeration type must satisfy the ordering relation of the type. The aggregate must be named when the enumeration type has a single literal.

Example:

```
type MIX_CODE is (ADD, SUB, MUL, LDA, STA, STZ);

for MIX_CODE use
  (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24, STZ => 33);
```

Notes:

The predefined attributes SUCC, PRED, and ORD are defined even for enumeration types with a non-contiguous representation. In this case, the functions are less efficiently implemented due to the need to avoid the omitted values. Similar considerations apply when such types are used for indexing.

13.4 Record Type Representations

A record type representation specifies the storage representation of records, that is, the order, position, and size of record components.

```
record_type_representation ::=
   for type_name use
      record [alignment_clause;]
         {component_name location;}
      end record;

location ::= at static_expression range range

alignment_clause ::= at mod static_expression
```

The position of a component is specified as a location relative to the start of the record; the **at** clause defines the address of a storage unit and the range defines the bit positions of the component relative to the storage unit.

The first storage unit of a record is numbered 0. The first bit of a storage unit is numbered 0. The ordering of bits in a storage unit is machine dependent and may extend to adjacent storage units. For a specific machine, the size in bits of a storage unit is given by the configuration dependent constant SYSTEM'STORAGE_UNIT.

Locations may be specified for some or for all components of a record. If no location is specified for a component, freedom is left to the compiler to define the location of the component. Locations within a record variant must not overlap, but the storage for distinct variants may overlap. Each location must allow for enough storage space to accommodate every allowable value of the component.

An alignment clause forces each record of the given type to be allocated at a start address which is a multiple of the value of the expression (i.e. the address modulo the expression must be zero). An implementation may place restrictions on the allowable alignments. Components may overlap storage boundaries, but an implementation may place restrictions on how components may overlap storage boundaries.

Examples:

```
WORD : constant INTEGER := 4;   -- storage unit is byte, 4 bytes per word

type STATE is (A, M, W, P);
type MODE is (FIX, DEC, EXP, SIGNIF);

type PROGRAM_STATUS_WORD is
   record
      SYSTEM_MASK       : array(0 .. 7) of BOOLEAN;
      PROTECTION_KEY    : INTEGER range 0 .. 3;
      MACHINE_STATE     : array(STATE'FIRST .. STATE'LAST) of BOOLEAN;
      INTERRUPT_CAUSE   : INTERRUPTION_CODE;
      ILC               : INTEGER range 0 .. 3;
      CC                : INTEGER range 0 .. 3;
      PROGRAM_MASK      : array(MODE'FIRST .. MODE'LAST) of BOOLEAN;
      INST_ADDRESS      : ADDRESS;
   end record;
```

```
for PROGRAM_STATUS_WORD use
  record at mod 8;
    SYSTEM_MASK       at 0*WORD range 0 .. 7;
    PROTECTION_KEY    at 0*WORD range 10 .. 11;   -- bits 8, 9 unused
    MACHINE_STATE     at 0*WORD range 12 .. 15;
    INTERRUPT_CAUSE   at 0*WORD range 16 .. 31;
    ILC               at 1*WORD range 0 .. 1;     -- second word
    CC                at 1*WORD range 2 .. 3;
    PROGRAM_MASK      at 1*WORD range 4 .. 7;
    INST_ADDRESS      at 1*WORD range 8 .. 31;
  end record;
```

13.5 Address Specifications

An address specification defines the location of an object in storage or the start address of a program unit.

address_specification ::= **for** name **use at** *static*_expression;

The **at** clause specifies an absolute address expressed in storage units in an implementation defined address space. The name must be one of the following:

(a) Name of a variable: in this case, the address is the address of the variable.

(b) Name of a subprogram or module: the address is that of the machine code associated with the subprogram or module.

(c) Name of an entry: The address is that of a hardware interrupt to which the entry is linked. The conventions defining the mapping between the integer value of the expression and the interrupt are implementation dependent.

13.5.1 Interrupts

An interrupt acts as an entry call. An accept statement for such an entry results in suspension of the task until the interrupt occurs. If control information is supplied by the interrupt, then this must appear as an **in** parameter of the entry. Multiple interrupts are queued on the corresponding entry. There may be an implementation defined limit for the number of allowed pending interrupts on a given entry.

Example of interrupt specification:

```
task CARD_READER_INTERRUPT is
  entry ATTENTION;
end;
```

```
task body CARD_READER_INTERRUPT is
   entry DONE;
   for DONE use at 4;
begin
   loop
      accept ATTENTION;
      select
         accept DONE;
         CARD_READER.EMPTY;
      or
         delay 2*SEC;
         CARD_READER.FINISH;
      end select;
   end loop;
end CARD_READER_INTERRUPT;
```

13.6 Change of Representations

Only one representation can be defined for a given type. In consequence if an alternative representation is desired, it is necessary to declare a second type derived from the first and to specify a different representation for the second type.

Example:

```
-- PACKED_DESCRIPTOR and DESCRIPTOR are two different types
-- with identical characteristics, apart from their representation

type DESCRIPTOR is
   record
      -- components of a descriptor
   end;

type PACKED_DESCRIPTOR is new DESCRIPTOR;

for PACKED_DESCRIPTOR use packing;
```

Change of representations can now be accomplished by assignment with explicit type conversions. Thus:

```
D  : DESCRIPTOR;
P  : PACKED_DESCRIPTOR;

P  := PACKED_DESCRIPTOR(D);   -- pack
D  := DESCRIPTOR(P);          -- unpack
```

13.7 Configuration and Machine Dependent Constants

The characteristics of the configuration can be specified by supplying appropriate pragmas:

```
pragma SYSTEM(name);              -- to establish the name of the object machine
pragma STORAGE_UNIT(number);      -- to establish the number of bits per storage unit
pragma MEMORY_SIZE(number);       -- to establish the available number of storage units
```

The values of configuration dependent constants are denoted by predefined attributes of the predefined name SYSTEM. Similarly, compiler options may be interrogated with boolean attributes of the predefined name OPTION.

```
SYSTEM'NAME              -- the name of the object machine
SYSTEM'STORAGE_UNIT      -- the number of bits per storage unit
SYSTEM'MEMORY_SIZE       -- the number of available storage units in memory

OPTION'SPACE             -- true if space is the optimization criterion
OPTION'TIME              -- true if time is the optimization criterion
```

Other implementation dependent characteristics of specific program constructs, including the characteristics established by representation specifications, can be determined using appropriate attributes. An implementation may provide additional predefined attributes specific to the machine considered. The list of the predefined attributes that are defined by the language is given in appendix A.

Examples:

```
INTEGER'SIZE      -- number of bits actually used for implementing INTEGER
TABLE'ADDRESS     -- the address of TABLE in storage units

X.COMPONENT'POSITION    -- position of COMPONENT in storage units
X.COMPONENT'FIRST_BIT   -- first bit of bit range
X.COMPONENT'LAST_BIT    -- last bit of bit range
```

13.8 Machine Code Insertions

A machine code insertion may be achieved by a call to an inline procedure whose body contains only code statements.

```
code_statement ::= qualified_expression;
```

Each machine instruction appears as a record aggregate of a record type defining the corresponding instruction. Declarations of such record types will generally be available in a predefined package for each machine. A procedure that contains a code statement must contain only code statements.

An implementation may provide machine dependent pragmas specifying register and calling conventions.

Example:

```
M: MASK;

procedure SET_MASK is
   use INSTRUCTION_360;
   pragma INLINE;
begin
   SI_FORMAT(CODE => SSM, B => M'BASE, D => M'DISP);
   -- M'BASE and M'DISP are implementation specific predefined attributes
end;
```

13.9 Interface to Other Languages

A subprogram written in another language can be called from a Green program provided that all communication is achieved via parameters and result. A procedure skeleton must be provided for such foreign subprograms. This skeleton must include a subprogram specification in the usual form, thus enabling other subprograms to call it. The sequence of statements of this skeleton must reduce to a pragma specifying the foreign language.

For example, the Fortran subprogram

```
SUBROUTINE GAUSS(A,X,N)
DIMENSION A(10,10), X(10)
 ...
END
```

can be represented by the following skeleton

```
type MATRIX  is array (INTEGER, INTEGER) of REAL;
type VECTOR  is array (INTEGER) of REAL;
   ...
procedure GAUSS(A : in out MATRIX; X : in out VECTOR; N : INTEGER) is
begin
   pragma INTERFACE(FORTRAN);
end;
```

The pragma specifies the calling convention to be used, and informs the compiler that an object module will be provided for the corresponding subprogram.

This capability need not be provided by all compilers; an implementation may place restrictions on the allowable forms and places of parameters and calls.

13.10 Unsafe Type Conversion

Unsafe type conversions can be achieved by program units having access to the predefined library module UNSAFE_PROGRAMMING by instantiating the generic function UNSAFE_CONVERSION.

```
package UNSAFE_PROGRAMMING is

   generic (type S ; type T)
   function UNSAFE_CONVERSION (X : S) return T;

end UNSAFE_PROGRAMMING;
```

It is a consequence of the visibility rules that such program units must include UNSAFE_PROGRAMMING in their visibility list.

14. Input-Output

This chapter describes the input-output facilities defined in the language. The standard generic package INPUT_OUTPUT defines a set of input-output primitives applicable to files containing elements of a single type. Additional primitives for text input-output are supplied in the standard package TEXT_IO. These facilities are described here as well as the conventions to be used for dealing with low level input-output operations.

14.1 General User Level Input-Output

The high level input-output facilities are defined in the language. A suitable package is described here and is given explicitly in section 14.2; it defines file types and the procedures and functions which operate on files.

Files are declared and associated with appropriate sources and destinations (called *external files*) such as peripheral devices or data sets. Distinct file types are defined to provide either read-only access, write-only access or read-and-write access to external files. The corresponding file types are called IN_FILE, OUT_FILE, and INOUT_FILE.

At this level, external files are named by a character string, which is interpreted by individual implementations to distinguish peripherals, access rights, physical organization, etc.

The package defining these facilities is generic and is called INPUT_OUTPUT. Any program which requires these facilities must instantiate the package for the appropriate element type.

A file can be read or written, and it can be set to a required position; the current position for access and the number of elements in the file may be obtained.

14.1.1 Files

A file is associated with an ordered collection of elements, all of the same type. The type of the elements is specified as a parameter in the package declaration, so that appropriate procedures are produced for dealing with elements of that type, as well as the file types. For example:

```
package INT_IO is new INPUT_OUTPUT(INTEGER);
```

establishes types and procedures for files of integers, so that

```
INT_FILE : INT_IO.OUT_FILE;
```

declares INT_FILE as a write-only file of integers.

Before any file processing can be carried out, the file must be associated with an external file. When such an association is in effect, the file is said to be open. This operation is performed by one of the following two procedures:

CREATE — establishes a new external file and associates the given file with it. If the given file is already open, the exception FILE_OPEN_ERROR is raised. If creation is prohibited for an external file because it already exists or for any other reason, the exception FILE_NAME_ERROR is raised. (CREATE is not defined for an IN_FILE.)

OPEN — associates the given file with an existing external file. If the given file is already open, the exception FILE_OPEN_ERROR is raised. If no such file exists or this access is prohibited, the exception FILE_NAME_ERROR is raised.

Any file operation that cannot be completed because of difficulties in the underlying system raises the exception MALFUNCTION. Any attempt to carry out an operation that is physically impossible or prohibited raises the exception FILE_USE_ERROR.

Whether a file is currently associated with an external file may be discovered by the following function:

IS_OPEN — returns TRUE if there is an associated file, FALSE otherwise.

All operations described subsequently apply to open files. Any attempt to use them on a file that is not open raises the FILE_OPEN_ERROR exception.

The name of the external file currently associated with a given file can be discovered by the function:

NAME — returns a string representing the name of the external file currently associated with the argument.

After processing has been completed on a file, the association may be broken by one of the following two procedures:

CLOSE — breaks the association between the file and its associated external file. The external file continues to exist.

DELETE — breaks the association between the file and its associated external file. The external file is deleted.

Example 1: Create a New External File on Backing Store

```
CREATE(FILE :=: INT_FILE, NAME := ">udd>ada>counts");
-- write the file
CLOSE(INT_FILE);
```

Example 2: Read a Paper Tape

```
declare
   package CHAR_IO is new INPUT_OUTPUT(CHARACTER);
   PT : CHAR_IO.IN_FILE;
begin
   CHAR_IO.OPEN(PT, "ttyg");
   -- input the file from device ttyg
   CHAR_IO.CLOSE(PT);
end;
```

14.1.2 File Processing

An open file has a current size and a current position. The size of a file is the total number of elements currently in it. The current position of the file determines the next element to be read or written. Each element occupies one position, counting from 1. After the last element of a file has been read or written, the next element position equals the size of the file plus one.

The following subprograms are available for input-output and for handling the size and position of a file:

READ — obtains the next element value from the external file and advances the position by one. If there is no such element then the exception END_OF_FILE is raised. If the next element is not of the proper type then the exception INVALID_DATA is raised. (READ is not defined for an OUT_FILE; for an INOUT_FILE, any previous WRITE must have been completed.)

WRITE — gives the next element in the external file the specified value and advances the position by one, adjusting the size of the file if necessary. (WRITE is not defined for an IN_FILE.)

SIZE — returns the number of elements currently in the file.

NEXT — returns the current position of the file.

SET_NEXT — moves the file so that the next element read or written will be from the position specified. If the position value specified does not exist in the current file, a subsequent READ will raise an exception (a subsequent WRITE may raise an exception). SET_NEXT has a default position corresponding to the first element of the file. This procedure can be used to rewind, backspace, or advance the file.

Examples of file positioning:

```
SET_NEXT(INT_FILE);                          -- rewind (set to position 1)
SET_NEXT(INT_FILE, NEXT(INT_FILE)-1);        -- backspace
SET_NEXT(INT_FILE, SIZE(INT_FILE)+1);        -- advance to end of file
```

Example of file processing:

```
-- Accumulate the values in a file and append the total

declare
   use INT_IO;
   COUNTS  : INOUT_FILE;
   VALUE   : INTEGER;
   TOTAL   : INTEGER := 0;
begin
   OPEN(COUNTS, ">udd>ada>counts");
   loop
      READ(COUNTS, VALUE);
      TOTAL := TOTAL + VALUE;
   end loop;
exception
   when END_OF_FILE =>
      WRITE (COUNTS, TOTAL);
      CLOSE (COUNTS);
end;
```

14.2 Specification of the Package INPUT OUTPUT

The specification of the generic standard package INPUT_OUTPUT is given below. It provides the calling conventions for all operations described in section 14.1.

```
generic (type ELEMENT_TYPE)
package INPUT_OUTPUT is
   restricted type IN_FILE     is private;
   restricted type OUT_FILE    is private;
   restricted type INOUT_FILE  is private;

-- Global operations for file manipulation

   procedure  CREATE (FILE : in out OUT_FILE;    NAME : in STRING);
   procedure  CREATE (FILE : in out INOUT_FILE;  NAME : in STRING);

   procedure  OPEN   (FILE : in out IN_FILE;     NAME : in STRING);
   procedure  OPEN   (FILE : in out OUT_FILE;    NAME : in STRING);
   procedure  OPEN   (FILE : in out INOUT_FILE;  NAME : in STRING);

   procedure  CLOSE  (FILE : in out IN_FILE);
   procedure  CLOSE  (FILE : in out OUT_FILE);
   procedure  CLOSE  (FILE : in out INOUT_FILE);

   function   IS_OPEN(FILE : in IN_FILE)  return BOOLEAN;
   function   IS_OPEN(FILE : in OUT_FILE) return BOOLEAN;
   function   IS_OPEN(FILE : in INOUT_FILE) return BOOLEAN;

   function   NAME   (FILE : in IN_FILE)  return STRING;
   function   NAME   (FILE : in OUT_FILE) return STRING;
   function   NAME   (FILE : in INOUT_FILE) return STRING;

   function   SIZE   (FILE : in IN_FILE)  return INTEGER;
   function   SIZE   (FILE : in OUT_FILE) return INTEGER;
   function   SIZE   (FILE : in INOUT_FILE) return INTEGER;
```

```
   -- input and output subprograms

      procedure  READ     (FILE : in IN_FILE;     ITEM : out ELEMENT_TYPE);
      procedure  READ     (FILE : in INOUT_FILE; ITEM : out ELEMENT_TYPE);

      procedure  WRITE    (FILE : in OUT_FILE;    ITEM : in ELEMENT_TYPE);
      procedure  WRITE    (FILE : in INOUT_FILE; ITEM : in ELEMENT_TYPE);

      function   NEXT     (FILE : in IN_FILE) return INTEGER;
      function   NEXT     (FILE : in OUT_FILE) return INTEGER;
      function   NEXT     (FILE : in INOUT_FILE) return INTEGER;

      procedure  SET_NEXT(FILE : in IN_FILE;     POS : in INTEGER := 1);
      procedure  SET_NEXT(FILE : in OUT_FILE;    POS : in INTEGER := 1);
      procedure  SET_NEXT(FILE : in INOUT_FILE; POS : in INTEGER := 1);

   -- exceptions that can be raised

      FILE_NAME_ERROR   : exception;
      FILE_USE_ERROR    : exception;
      FILE_OPEN_ERROR   : exception;
      INVALID_DATA      : exception;
      MALFUNCTION       : exception;
      END_OF_FILE       : exception;

   private
      -- declarations of the file private types
   end INPUT_OUTPUT;
```

14.3 Text Input-Output

Facilities are available for input and output in human readable form, with the external file consisting of characters. These are provided in a package given in section 14.4 and explained here. This package defines character file types and the procedures and functions which put and get values of objects in and out of such files.

The package defining these facilities is called TEXT_IO. It uses the general INPUT_OUTPUT package for files of type CHARACTER, so that all the facilities described in section 14.1 are available. In addition to these general facilities, procedures are provided to GET and PUT values of suitable types, carrying out conversions between the internal values and appropriate character strings.

All the GET and PUT procedures have an ITEM parameter whose type determines the details of the action and the appropriate character string in the external file. Note that the ITEM parameter is an *out* parameter in GET and an *in* parameter for PUT. The general principle is that the characters in the external file are composed and analyzed as lexical elements, as described in chapter 2. The conversions are based on the REP and VAL attributes described in Appendix A.

For all GET and PUT procedures, there are forms with and without a file specified. If a file is specified, it must be of the correct type (IN_FILE for GET, OUT_FILE for PUT). If no file is specified, a default input or output file is used. At the beginning of program execution, the default input and output files are the so-called standard input and output files, which are associated with two implementation defined external files.

14.3.1 Standard Input and Output Files

The particular files used as default for the short forms of GET and PUT can be manipulated by the following functions and procedures:

```
function STANDARD_INPUT  return IN_FILE;     -- returns initial default in-file.
function STANDARD_OUTPUT return OUT_FILE;    -- returns initial default out-file.
procedure SET_INPUT  (FILE : in IN_FILE);    -- sets the default in-file
procedure SET_OUTPUT (FILE : in OUT_FILE);   -- sets the default out-file
```

14.3.2 Layout

The characters in the file are considered to form a sequence of lines. The characters in each line are considered to occupy consecutive columns, counting from 1. The lines in each file are counted from 1. A file may have a particular line length that is explicitly set by the user. If no line length has been specified, lines can be of any length up to the size of the file. During file processing, a file has a current line number and a current column number. These determine the starting position available for the next GET or PUT operation.

The characters in the file consist of printable (graphic) characters and control characters. Each printable character or space is associated with one column. The control characters have specific implications on the line and column numbers:

CR resets the column number to one.

LF increments the line number by one.

BS decrements the column number by one if it is greater than one.

TAB increments the column number to the next multiple of 8 unless that would take it beyond the line length or the end of line in an input file, in which case it increments only to the end of the line.

Other control characters do not affect the column number or line number.

Layout primitives manipulate the line structure of the file specified by the first parameter.

LINE	returns the current line number
COL	returns the current column number
SET_LINE	sets the current line to the value specified by the second parameter. The current column is set equal to 1.
SET_COL	sets the current column to the value specfied by the second parameter. The current line is unaffected.
SET_LINE_LENGTH	sets the line length to the value specified by the second parameter. Subsequently, if a SET_COL operation causes the column number to exceed the specified line length, the exception INVALID_LAYOUT is raised.

In addition, the predefined string NEWLINE corresponds to the sequence of characters used to denote the end of a line in a given implementation.

Examples:

```
SET_LINE(F, LINE(F)+1); -- advances to the next line
SET_COL(F, 20); -- advances (or backs up) to column 20 on the current line
SET_COL(F, COL(F) + 10); -- advances 10 columns forward
SET_LINE_LENGTH(F, 132);
PUT(NEWLINE); -- starts a new line on standard output
```

14.3.3 Input-Output of Characters and Strings

The GET and PUT procedures for these types work with individual characters, which may be printable or control characters, and affect the current column and line number in accordance with the particular characters processed (never going outside the permitted range).

For an ITEM of type CHARACTER

GET returns the value of the next character from the input file. If the line length is not fixed, this is the character corresponding to the current position on the file. If the line length is fixed, end of line marks are skipped and the next character is the one on the current column, or on the first column of the next line if the last column read corresponded to an end of line.

PUT outputs the character given as argument to the file. If the line length is not fixed, the character is output on the current column of the current line and the current column is updated according to the rules given in 14.3.2 above. If the line length is fixed, the character is output similarly and the current column is also updated according to the above rules, except if such a modification would cause the column number to become larger than the line length. In the latter case the line number is incremented by 1 and the column number is reset to 1 (an automatic new line is inserted). If the file cannot accept the character, the exception FILE_USE_ERROR is raised.

When the ITEM type is a string, the length of the string is determined and that exact number of GET or PUT operations for individual characters is carried out.

Example 1: *variable line length*:

```
PUT(F, "01234567" & NEWLINE & "89012345");
```

will output

```
01234567
89012345
```

If the file is subsequently read, the whole string can be obtained by

```
X : STRING(1 .. 18);
  ...
GET(F, X);
```

(assuming NEWLINE is the string CR & LF, i.e., two characters).

Example 2: fixed line length of 8:

```
PUT(F, "0123456789012345");
```

will output

```
01234567
89012345
```

A subsequent read of that text, with the same line length, could be performed by

```
X : STRING(1 .. 16);
  ...
GET(F, X);
```

Note that the double-quote marks enclosing an actual parameter to PUT are not output, but the string inside is output with any doubled double-quote marks written once, thus matching the rule for character strings (see 2.5).

14.3.4 Input-Output for Other Types

All ITEM types other than CHARACTER or STRING are treated in a uniform way, as lexical units (see 2.2, 2.3, 2.4). The output is a character string having the syntax described for the appropriate unit and the input is taken as the longest possible character string having the required syntax. For input, any leading space, TAB, CR, or LF characters are ignored. A consequence is that no such units can cross a line boundary, and the line number is not changed by them.

If the character string read is not consistent with the syntax of the required lexical unit, the exception INVALID_DATA is raised.

The PUT procedures for numeric and enumeration types include an optional WIDTH parameter, which specifies a minimum number of characters to be generated. If the width given is larger than the string representation of the value, the value will be preceded (for numeric types) or followed (for enumeration type) by the appropriate number of spaces. If the field width is smaller than the string representation of the value, the field width is ignored. In all cases, the string printed is preceded by a space, except at the first column of a line. A default width of 0 is provided, thus giving the minimum number of characters.

In each PUT operation, if the line can accommodate all the characters generated, then the characters are placed on that line from the current column. If the line cannot accommodate all the characters, then a new line is started and the characters are placed on the new line starting from column 1.

14.3.5 Input-Output for Numeric Types

Integers:

GET reads an optional minus or plus sign then according to the syntax of integer. If the value obtained is outside the implemented range of integer numbers, the exception OVERFLOW is raised.

PUT expresses the ITEM value as a decimal integer, with no underscores and no leading zeros (but a single 0 for the value zero) and a preceding minus sign for a negative value; this string is right justified in the field width and padded with spaces.

Floating point numbers:

GET reads an optional plus or minus sign then according to the syntax of approximate numbers. The value obtained is rounded to the precision FLOAT'DIGITS. If the value obtained is outside the implemented range for type FLOAT, the exception OVERFLOW is raised.

PUT expresses the ITEM value as the nearest approximate number with one digit before the decimal point, the specified number of digits after the decimal point, and an exponent part with a sign and three digits. If the specified number of decimals is not positive, then the decimal point and fractional part are omitted. If the specified number of decimals is smaller than that authorized by the precision, rounding is performed.

Fixed point numbers:

Because the representation of a fixed point type cannot be predetermined, these procedures are contained in a generic package.

GET reads an optional plus or minus sign then according to the syntax of an approximate number. The value obtained is rounded to the appropriate delta.

PUT expresses the ITEM value as the nearest approximate number with the specified number of digits after the decimal point. A default is provided to accommodate the delta. If the specified number of digits is smaller than that needed to represent delta, then rounding is performed.

14.3.6 Input-Output for Boolean

GET reads an identifier according to the syntax given in 2.3, with no distinction between upper and lower case letters. If the identifier is TRUE or FALSE, then the boolean value is given; otherwise the exception INVALID_DATA is raised.

PUT expresses the ITEM value as TRUE or FALSE and puts these letters in upper case.

14.3.7 Input-Output for Enumeration Types

Because each enumeration type has its own set of literals, these procedures are contained in a generic package. An instantiation must specify the type.

GET reads an identifier (according to the syntax given in 2.3, with no distinction between upper and lower case letters) or a character literal (according to the syntax of 2.5 for a single character in double-quotes). If this is one of the enumeration literals of the type, then the enumeration value is given; otherwise the exception INVALID_DATA is raised.

PUT outputs the ITEM value as an identifier in upper case or as a character literal.

14.4 Specification of the Package TEXT IO

The package TEXT_IO contains the definition of all the text input-output primitives.

```
package TEXT_IO is
   package CHARACTER_IO is new INPUT_OUTPUT(CHARACTER);
   type IN_FILE     is new CHARACTER_IO.IN_FILE;
   type OUT_FILE    is new CHARACTER_IO.OUT_FILE;

-- Character Input-Output

   procedure GET ( FILE   : in   IN_FILE; ITEM : out CHARACTER);
   procedure GET ( ITEM   : out  CHARACTER);
   procedure PUT ( FILE   : in   OUT_FILE; ITEM : in CHARACTER);
   procedure PUT ( ITEM   : in   CHARACTER);

-- String Input-Output

   procedure GET ( FILE   : in   IN_FILE; ITEM : out STRING);
   procedure GET ( ITEM   : out  STRING);
   procedure PUT ( FILE   : in   OUT_FILE; ITEM : in STRING);
   procedure PUT ( ITEM   : in   STRING);

-- Integer Input-Output

   DEFAULT_WIDTH : constant INTEGER := 0;
   procedure GET ( FILE     : in   IN_FILE; ITEM : out INTEGER);
   procedure GET ( ITEM     : out  INTEGER);
   procedure PUT ( FILE     : in   OUT_FILE;
                   ITEM     : in   INTEGER;
                   WIDTH    : in   INTEGER := DEFAULT_WIDTH);
   procedure PUT ( ITEM     : in   INTEGER;
                   WIDTH    : in   INTEGER := DEFAULT_WIDTH);
```

```
-- Floating Point Input-Output

   DEFAULT_FLOAT_WIDTH  : constant INTEGER := 0;
   DEFAULT_MANTISSA     : constant INTEGER := FLOAT'DIGITS - 1;
   procedure GET ( FILE     : in  IN_FILE; ITEM : out FLOAT);
   procedure GET ( ITEM     : out FLOAT);
   procedure PUT ( FILE     : in  OUT_FILE;
                   ITEM     : in  FLOAT;
                   WIDTH    : in  INTEGER := DEFAULT_FLOAT_WIDTH;
                   FRACT    : in  INTEGER := DEFAULT_MANTISSA);
   procedure PUT ( ITEM     : in  FLOAT;
                   WIDTH    : in  INTEGER := DEFAULT_FLOAT_WIDTH;
                   FRACT    : in  INTEGER := DEFAULT_MANTISSA);

-- Fixed Point Input-Output is defined as a generic package

   generic( type FIXED_TYPE;
            function REP(X : FIXED_TYPE) return STRING is FIXED_TYPE'REP;
            function VAL(X : STRING) return FIXED_TYPE is FIXED_TYPE'VAL;
            DECIMALS : FIXED_TYPE := FIXED_TYPE'DELTA)
   package FIXED_IO is
      DEFAULT_FIXED_WIDTH : constant INTEGER := 0;
      DELTA_REP           : constant STRING := REP(DECIMALS - INTEGER(DECIMALS));
      DEFAULT_DECIMALS    : constant INTEGER := DELTA_REP'LENGTH - 2;
      procedure GET ( FILE     : in  IN_FILE; ITEM : out FIXED_TYPE);
      procedure GET ( ITEM     : out FIXED_TYPE);
      procedure PUT ( FILE     : in  OUT_FILE;
                      ITEM     : in  FIXED_TYPE;
                      WIDTH    : in  INTEGER := DEFAULT_FIXED_WIDTH;
                      FRACT    : in  INTEGER := DEFAULT_DECIMALS);
      procedure PUT ( ITEM     : in FIXED_TYPE;
                      WIDTH    : in INTEGER := DEFAULT_FIXED_WIDTH;
                      FRACT    : in INTEGER := DEFAULT_DECIMALS);
   end FIXED_IO;

-- Boolean Input-Output

   DEFAULT_ENUM_WIDTH : constant INTEGER := 0;
   procedure GET ( FILE     : in  IN_FILE; ITEM : out BOOLEAN);
   procedure GET ( ITEM     : out BOOLEAN);
   procedure PUT ( FILE     : in  OUT_FILE;
                   ITEM     : in  BOOLEAN;
                   WIDTH    : in  INTEGER := DEFAULT_ENUM_WIDTH);
   procedure PUT ( ITEM     : in  BOOLEAN;
                   WIDTH    : in  INTEGER := DEFAULT_ENUM_WIDTH);
```

```
-- Generic package for enumeration types

   generic(type ENUM_TYPE;
           function REP(X : ENUM_TYPE) return STRING is ENUM_TYPE'REP;
           function VAL(X : STRING) return ENUM_TYPE is ENUM_TYPE'VAL)
   package ENUM_IO is
      procedure GET ( FILE     : in  IN_FILE; ITEM : out ENUM_TYPE);
      procedure GET ( ITEM     : out ENUM_TYPE);
      procedure PUT ( FILE     : in  OUT_FILE;
                      ITEM     : in  ENUM_TYPE;
                      WIDTH    : in  INTEGER := DEFAULT_ENUM_WIDTH);
      procedure PUT ( ITEM     : in  ENUM_TYPE;
                      WIDTH    : in  INTEGER := DEFAULT_ENUM_WIDTH);
   end ENUM_IO;

-- Layout primitives

   function LINE(FILE : in IN_FILE)    return NATURAL;
   function LINE(FILE : in OUT_FILE)   return NATURAL;

   function COL(FILE : in IN_FILE)     return NATURAL;
   function COL(FILE : in OUT_FILE)    return NATURAL;

   procedure SET_LINE(FILE : in IN_FILE;  TO : in NATURAL);
   procedure SET_LINE(FILE : in OUT_FILE; TO : in NATURAL);

   procedure SET_COL(FILE : in IN_FILE;   TO : in NATURAL);
   procedure SET_COL(FILE : in OUT_FILE;  TO : in NATURAL);

   procedure SET_LINE_LENGTH(FILE : in IN_FILE;  N : in NATURAL);
   procedure SET_LINE_LENGTH(FILE : in OUT_FILE; N : in NATURAL);

   NEWLINE : constant STRING := implementation defined;

-- Standard input and output file manipulation

   function STANDARD_INPUT  return IN_FILE;
   function STANDARD_OUTPUT return OUT_FILE;

   procedure SET_INPUT  (FILE : in IN_FILE);
   procedure SET_OUTPUT (FILE : in OUT_FILE);

-- Exceptions

   FILE_NAME_ERROR  : exception;
   FILE_USE_ERROR   : exception;
   FILE_OPEN_ERROR  : exception;
   INVALID_DATA     : exception;
   MALFUNCTION      : exception;
   END_OF_FILE      : exception;
   INVALID_LAYOUT   : exception;

end TEXT_IO;
```

14.5 Example of Text Input-Output

The following example shows the use of the text input-output primitives in a dialogue with a user at a terminal. The user is asked to select a color, and the program output in response is the number of items of the color available in stock. The default input and output files are used.

```
type COLOR is (WHITE, RED, ORANGE, YELLOW, GREEN, BLUE, BROWN);
TARGET : array (WHITE .. BROWN) of INTEGER := (20, 17, 43, 10, 28, 173, 87);

package COLOR_IO is new ENUM_IO(COLOR);

procedure DIALOGUE is
  use COLOR_IO;
  CHOICE: COLOR;

  procedure READLN is
    CH : CHARACTER;
  begin
    loop
      GET(CH);
      exit when CH = LF;
    end loop;
  end;

  procedure ENTER_COLOR return COLOR is
    CHOICE : COLOR;
  begin
    loop
      begin
        PUT("Color selected: ");
        GET(CHOICE);
        READLN;
        return CHOICE;
      exception
        when INVALID_DATA =>
          READLN;
          PUT("Invalid color, try again.");
      end;
    end loop;
  end;
begin -- body of DIALOGUE
  CHOICE := ENTER_COLOR;
  PUT(NEWLINE);
  PUT(CHOICE); PUT("items available:");
  SET_COL(STANDARD_OUTPUT, 25);
  PUT(TARGET(CHOICE), WIDTH := 5);
  PUT(";" & NEWLINE);
end DIALOGUE;
```

Example of an interaction (characters typed by the user are italicized):

```
Color selected: black
Invalid color, try again. Color selected: blue
BLUE items available:      173;
```

14.6 Low Level Input-Output

A low level input-output operation is an operation acting on a physical device. Such an operation is handled by using one of the (overloaded) predefined procedures SEND_CONTROL and RECEIVE_-CONTROL.

A procedure SEND_CONTROL may be used to send control information to a physical device. A procedure RECEIVE_CONTROL may be used to monitor the execution of an input-output operation by requesting information from the physical device.

Such procedures are declared in the standard package LOW_LEVEL_IO and have two parameters identifying the device and the data. However, the kinds and formats of the control information will depend on the physical characteristics of the machine and the device. Hence the types of the parameters are implementation defined. Overloaded definitions of these procedures should be provided for the supported devices.

The visible part of the package defining these procedures is outlined as follows:

```
package LOW_LEVEL_IO is
   -- declarations of the possible types for DEVICE and DATA
   -- declarations of overloaded procedures for these types:
   procedure SEND_CONTROL    (DEVICE : device_type; DATA : in out data_type);
   procedure RECEIVE_CONTROL (DEVICE : device_type; DATA : in out data_type);
end;
```

The bodies of the procedures SEND_CONTROL and RECEIVE_CONTROL for various devices can be supplied in the body of the package LOW_LEVEL_IO. These procedure bodies may be written with code statements.

A. Predefined Language Attributes

The following attributes are predefined in the language and are denoted by the notation described in section 4.1, where the name of the entity is followed by an apostrophe and the attribute identifier.

Attributes of any object or subprogram

ADDRESS — X'ADDRESS returns an integer corresponding to the location of the first storage cell of X, which is given as an index in a linear memory.

Attributes of any type or subtype (or objects thereof)

SIZE — Gives the number of bits used to implement objects of the type.

Attributes of any scalar type or subtype

FIRST — T'FIRST returns the minimum value in the range of T.

LAST — T'LAST returns the maximum value in the range of T.

REP(X) — X being an object, T'REP(X) returns the string representation of the value of X. The string contains the minimum number of characters needed, i.e.:

- For a number, only significant digits are given, and the number of decimal digits is the smallest number required to express the mantissa within the declared accuracy. A floating point number is represented in scientific notation with one digit before the decimal point, and a signed three digit exponent. A fixed point number is represented in fixed point notation. All numbers are expressed in base 10.
- For enumeration literals, identifiers are represented in uppercase, without any extra space. Character values are represented surrounded by double quotes.

VAL(X) — X being a string, T'VAL(X) returns the value of type T whose external representation is given by the string X. If no interpretation of X can be given within T, the RANGE_ERROR exception is raised.

Attributes of any real type or subtype

BITS — Returns an integer indicating the minimum number of bits needed to represent the mantissa of the type or subtype.

Attributes of any discrete type or subtype T

PRED(X)	Returns the value preceding X in T. The exception RANGE_ERROR is raised if X = T'FIRST.
SUCC(X)	Returns the value following X in T. The exception RANGE_ERROR is raised if X = T'LAST.
ORD(X)	Returns an integer which is the ordinal position of X in the type or subtype T. Note that T'ORD(T'FIRST) = 1, and that ORD is independent of any representation specification: T'ORD(T'SUCC(X)) = T'ORD(X) + 1.
VAL(I)	I being an integer, T'VAL(I) returns the enumeration value occupying the I-th position in T. The exception RANGE_ERROR is raised if I is not in the range T'ORD(T'FIRST) .. T'ORD(T'LAST). For any enumeration value X of type T, T'VAL(T'ORD(X)) = X.

Attributes of any numeric type or subtype

RADIX	Returns an integer giving the actual radix (base) used to represent values of the type.

Attributes of any fixed point type or subtype T

DELTA	Returns a fixed point number indicating the value of the delta specified in the type declaration.
SMALL	Returns a fixed point number indicating the closest (positive or negative) power of 2 immediately inferior to the delta. For example, if T'DELTA = 0.005, then T'SMALL is 0.00390625 (1/256).
LARGE	Returns a fixed point number indicating the largest permissible value that can be expressed in the binary representation of T, with a delta of T'SMALL: if T is declared "**delta** 0.01 **range** -100.0 .. 100.0" then T'LARGE = 256 - 1/128 = 255.9921875.

Attributes of any floating point type or subtype

DIGITS	Returns an integer indicating the number of decimal digits specified in the type or subtype declaration.
LARGE	Returns the largest positive value that can be expressed in the representation within the constraints of the precision of T.
SMALL	Returns the smallest positive value that can be expressed in the representation within the constraints of the precision of T.
EXPONENT_MIN	The minimum exponent (as an integer) possible in the representation used for values of the type.
EXPONENT_MAX	The maximum exponent (as an integer) possible in the representation used for values of the type.

Attributes of any array object and of any array type with specified bounds

FIRST	Returns a value in the type of the first index, which is the lower bound of that index.
FIRST(i)	Same as FIRST, for the i-th index.
LAST	Upper bound of the first index
LAST(i)	Same as LAST, for the i-th index.
LENGTH	Number of elements of the first dimension.
LENGTH(i)	Same as LENGTH, for the i-th dimension.

Attributes of any access type (or object thereof)

ACCESS_SIZE	Returns an integer giving the size in bits required for an object of the access type (whereas SIZE will give the size of the value denoted by such object).

Attributes of any record component

FIRST_BIT	Gives the bit position (as an integer) of the first bit of the component in the first storage unit used for the component. The first bit in the storage unit has the position 0.
LAST_BIT	Bit position of the last bit of the component from the beginning of the first storage unit used for the component.
POSITION	Gives the position (in storage units) of the first storage unit of the component relative to the beginning of the record. Position of the first component is 0.

Attributes of any task

ACTIVE	True if the corresponding task has been initiated, and is not yet terminated.
CLOCK	Returns a value of type TIME which is the cumulative processing time of the task since its last initiation. Can be used only in the task body.
INDEX	(for a task family) Returns the index (of the type of the family index) of the current member of the family. Can be used only in the task body.
PRIORITY	Returns a value of the predefined type PRIORITY, which is the current priority of the task. Can be used only in the task body.

Attributes of any entry

COUNT	Indicates the number of tasks currently awaiting a rendezvous with the entry. Can be used only within the body of the task containing the entry declaration.

Attributes of SYSTEM

CLOCK	Current time (of type TIME) indicated by the real time clock of the system.
MEMORY_SIZE	Number of storage-units available to the program.
NAME	Returns a literal of an implementation-defined enumeration type, identifying the target system.
STORAGE_UNIT	Number of bits in a storage unit.
MAX_PRIORITY	The maximum priority level for tasks on the system (of the predefined subtype PRIORITY).
MIN_PRIORITY	The minimum priority level.
MAX_INT	The maximum integer value supported by the machine.
MIN_INT	The minimum integer value supported by the machine.

Translator options

OPTION'SPACE	True if major optimization criterion is space efficiency.
OPTION'TIME	True if major optimization criterion is time efficiency.

B. Predefined Language Pragmas

Pragma Name	*Meaning*
CREATION	The identifier STATIC or DYNAMIC must be given as a parameter. The pragma must appear in the visible part of a task module. It specifies whether storage for the task is to be allocated statically or dynamically.
ENVIRONMENT	Takes a list of module names as arguments. The pragma must appear before a compilation unit. It specifies other visible modules to be included in the standard environment for all compilation units in the program library.
INCLUDE	Takes a string as argument, which is the name of a text file. The pragma can appear in a declarative part or sequence of statements. It specifies that the text file is to be included where the pragma is given.
INLINE	No arguments. The pragma must appear in the declarative part of a subprogram body. It specifies that the subprogram body should be expanded in line at each call.
INTERFACE	Takes an identifier as argument, indicating that the body of a subprogram is written in another language, whose linkage convention is to be observed. The list of possible languages is implementation defined.
LIST	The argument is either ON or OFF. The pragma can appear in a declarative part or sequence of statements. It specifies that listing of the program unit is to be continued or suspended until a LIST pragma is given with the opposite argument.
MEMORY_SIZE	Takes an integer argument indicating the number of storage units available in memory.
OPTIMIZE	The argument is TIME or SPACE. The pragma can appear before a compilation unit or within a declarative part or sequence of statements. It specifies whether time or space is the primary optimization criterion. The effect of the pragma holds until the end of the unit, unless another OPTIMIZE pragma overriding it is given.
PAGE	Indicates that the listing should start on a new page.
STORAGE_UNIT	Takes an integer argument indicating the number of bits to be used for each storage unit.
SUPPRESS	Exception names are given as arguments. The pragma must appear in the declarative part of a program unit. It specifies that within the unit no run-time checks need be provided for the named exceptions.
SYSTEM	An identifier is given as argument, and interpreted as the name of the target system of the program. The list of possible names is implementation defined.

C. Predefined Language Environment

This appendix outlines a definition of the predefined identifiers in the language. The definition is given in the form of a package module called STANDARD, which is implicitly inherited by all program units (see 8.6).

The types and other facilities given in STANDARD are deliberately minimal, since further packages, for instance, for complex arithmetic, can be expressed in the language itself. An implementation may, of course, make such packages a part of the standard environment. Furthermore, the predefined functions, operators, and procedures can be implemented in special ways that are particularly efficient on a given machine. For this reason, the corresponding package body is not shown.

Not all the predefined entities can be completely described in the language itself. For instance, although the enumeration type BOOLEAN can be written showing the two literals FALSE and TRUE, the relationship of BOOLEAN to conditions cannot be expressed in the language. In consequence, the text of the package given below does not give the complete semantics of the entities defined.

```
package STANDARD is

   type BOOLEAN is (FALSE, TRUE);

   function "not" (X : BOOLEAN) return BOOLEAN;

   function "and" (X,Y : BOOLEAN) return BOOLEAN;
   function "or"  (X,Y : BOOLEAN) return BOOLEAN;
   function "xor" (X,Y : BOOLEAN) return BOOLEAN;


   type SHORT_INTEGER  is range implementation_defined;
   type INTEGER        is range implementation_defined;
   type LONG_INTEGER   is range implementation_defined;

   function "+"   (X : INTEGER) return INTEGER;
   function "-"   (X : INTEGER) return INTEGER;
   function ABS   (X : INTEGER) return INTEGER;

   function "+"   (X,Y : INTEGER) return INTEGER;
   function "-"   (X,Y : INTEGER) return INTEGER;
   function "*"   (X,Y : INTEGER) return INTEGER;
   function "/"   (X,Y : INTEGER) return INTEGER;
   function "mod" (X,Y : INTEGER) return INTEGER;
   function "**"  (X : INTEGER; Y : INTEGER range 0 .. INTEGER'LAST) return INTEGER;

      -- Similarly for SHORT_INTEGER and LONG_INTEGER
```

```
type SHORT_FLOAT  is digits implementation_defined range implementation_defined;
type FLOAT        is digits implementation_defined range implementation_defined;
type LONG_FLOAT   is digits implementation_defined range implementation_defined;

function "+"  (X : FLOAT) return FLOAT;
function "-"  (X : FLOAT) return FLOAT;
function ABS  (X : FLOAT) return FLOAT;

function "+"  (X,Y : FLOAT) return FLOAT;
function "-"  (X,Y : FLOAT) return FLOAT;
function "*"  (X,Y : FLOAT) return FLOAT
function "/"  (X,Y : FLOAT) return FLOAT;
function "**" (X : FLOAT; Y : INTEGER) return FLOAT;

   -- Similarly for SHORT_FLOAT and LONG_FLOAT

-- The following characters comprise the standard ASCII character set.

type CHARACTER is
   ( NUL,   SOH,   STX,   ETX,   EOT,   ENQ,   ACK,   BEL,
     BS,    HT,    LF,    VT,    FF,    CR,    SO,    SI,
     DLE,   DC1,   DC2,   DC3,   DC4,   NAK,   SYN,   ETB,
     CAN,   EM,    SUB,   ESC,   FS,    GS,    RS,    US,
     " ",   "!",   """",  "#",   "$",   "%%",  "&",   "'",
     "(",   ")",   "*",   "+",   ",",   "-",   ".",   "/",
     "0",   "1",   "2",   "3",   "4",   "5",   "6",   "7",
     "8",   "9",   ":",   ";",   "<",   "=",   ">",   "?",
     "@",   "A",   "B",   "C",   "D",   "E",   "F",   "G",
     "H",   "I",   "J",   "K",   "L",   "M",   "N",   "O",
     "P",   "Q",   "R",   "S",   "T",   "U",   "V",   "W",
     "X",   "Y",   "Z",   "[",   "\",   "]",   "^",   "_",
     "`",   "a",   "b",   "c",   "d",   "e",   "f",   "g",
     "h",   "i",   "j",   "k",   "l",   "m",   "n",   "o",
     "p",   "q",   "r",   "s",   "t",   "u",   "v",   "w",
     "x",   "y",   "z",   "{",   "|",   "}",   "~",   DEL);

-- Enumeration literals for characters not in the basic language character set

EXCLAM      : constant CHARACTER := "!";
DOLLAR      : constant CHARACTER := "$";
QUESTION    : constant CHARACTER := "?";
AT_SIGN     : constant CHARACTER := "@";
L_BRACKET   : constant CHARACTER := "[";
BACK_SLASH  : constant CHARACTER := "\";
R_BRACKET   : constant CHARACTER := "]";
CIRCUMFLEX  : constant CHARACTER := "^";
GRAVE       : constant CHARACTER := "`";
LC_A        : constant CHARACTER := "a";
LC_B        : constant CHARACTER := "b";
  ...
LC_Z        : constant CHARACTER := "z";
L_BRACE     : constant CHARACTER := "{";
R_BRACE     : constant CHARACTER := "}";
TILDE       : constant CHARACTER := "~";
```

```
    subtype NATURAL is INTEGER range 1 .. INTEGER'LAST;
    type STRING is array (NATURAL) of CHARACTER;


    type TIME is digits implementation_defined range implementation_defined;
    SECONDS : constant TIME := implementation_defined;


    SYSTEM :  -- predefined name
    OPTION :  -- predefined name

    subtype PRIORITY is INTEGER range implementation_defined;
    procedure SET_PRIORITY(P : PRIORITY);


  -- Alphabetical list of exceptions

    ACCESS_ERROR          : exception;
    ASSERT_ERROR          : exception;
    DISCRIMINANT_ERROR    : exception;
    DIVIDE_ERROR          : exception;
    FAILURE               : exception;
    INITIATE_ERROR        : exception;
    NO_VALUE_ERROR        : exception;
    OVERFLOW              : exception;
    OVERLAP_ERROR         : exception;
    RANGE_ERROR           : exception;
    SELECT_ERROR          : exception;
    STORAGE_OVERFLOW      : exception;
    TASKING_ERROR         : exception;
    UNDERFLOW             : exception;

    generic task SEMAPHORE is
      entry P;
      entry V;
    end;

    generic task SIGNAL is
      entry SEND;
      entry WAIT;
    end;

    for CHARACTER use  -- 128 ASCII character set without holes
      (0, 1, 2, 3, 4, 5, 6, 7, 8,  ... , 126, 127);

    for STRING use packing;

  end STANDARD;
```

D. Glossary

Access type An access type is a type whose objects are allocated dynamically during program execution. An *access value* designates a dynamically allocated object.

Aggregate An aggregate is the written form denoting a value of a composite type. An *array aggregate* denotes a value for an array; a *record aggregate* denotes a value for a record.

Allocator An allocator creates a new dynamically allocated object and returns an access value designating the object.

Attribute An attribute is a characteristic of a named entity. A *predefined attribute* relates to the program entity, and is obtained by using one of the predefined attribute identifiers in the language. A *user defined attribute* exists only for types, and denotes a user defined subprogram for the type.

Body A body is a program unit defining the execution of a subprogram or module. A *body stub* is a replacement for a body that is compiled separately.

Compilation unit A compilation unit is a restricted program unit that is presented for compilation as a separate text. It may be a subprogram body, a module specification, or a module body.

Component A component denotes a part of a composite object or a nameable entity declared in a program unit. An *indexed component* is a name containing expressions denoting indices. An indexed component can denote either a component of an array, a task in a task family, or an entry in an entry family. A *selected component* is a name whose prefix names another declared entity and whose suffix denotes a component of the entity. A selected component can denote either a component of a record, an entity declared in another program unit, or a user defined type attribute.

Constraint A constraint is a restriction on the set of possible values of a type. A *range constraint* specifies the lower and upper bounds for the values of a scalar type. An *accuracy constraint* specifies the relative or absolute bounds on errors for a real type. An *index constraint* specifies the lower and upper bounds of an array index. A *discriminant constraint* specifies a particular variant for a record type.

Declarative part A declarative part is a sequence of declarations and related information such as subprogram bodies and representation specifications that apply over a region of program text.

Derived type A derived type is a type whose operations and values are taken from an existing type.

Discrete type The discrete types are the enumeration types and integer types. Discrete types may be used for indexing and iteration over loops.

Discriminant A discriminant is a constant component of a record. The value of the discriminant determines the particular variant of a given record value or the size of an array component.

Elaboration Elaboration is the process by which a declaration achieves its effect. For example it can associate a name with a program entity or initialize a newly declared variable.

Entry An entry is used for communication between tasks. Externally an entry is called just as a subprogram is called; its internal behavior is specified by one or more accept statements specifying the actions to be performed when the entry is called.

Exception An exception is an event that causes suspension of normal program execution. Bringing an exception to attention is called *raising* the exception. An *exception handler* is a piece of program text specifying a response to the exception. Execution of such a program text is called *handling* the exception.

Generic program unit A generic program unit is a subprogram or module specified with a generic clause. A *generic clause* contains the declaration of generic parameters. A generic program unit may be thought of as a possibly parameterized model of program units. Instances (i.e. filled-in copies) of the model can be obtained by *generic instantiation*. Such instantiated program units define subprograms and modules that can be used directly in a program.

Lexical unit A lexical unit is one of the basic syntactic elements comprising a program. A lexical unit is either an identifier, number, string, or delimiter.

Literal A literal denotes an explicit value of a given type, for example a number or a character string.

Module A module is a program unit specifying a group of logically related entities. The *visible part* of the module specification defines the information that another program unit is able to know about the module. The *private part* of the module specification defines the physical characteristics of the information specified in the visible part. A *module body* contains the bodies of specified subprograms and local declarative information, as well as statements to be executed when the module body is elaborated (for a package module) or initiated (for a task module).

Object An object is a variable or a constant. An object can denote any kind of data element, whether a scalar value, a composite value, or a value in an access type.

Overloading Overloading is the property of literals, identifiers, and operators that can have several alternative meanings within the same scope. For example an overloaded enumeration literal is a literal appearing in two or more enumeration types; an overloaded subprogram is a subprogram whose name can denote one of several subprogram bodies, depending upon the kind of its parameters and returned value.

Package A package is a module specifying a collection of related entities such as constants, variables, types, and subprograms.

Parameter A parameter is one of the named entities associated with a subprogram, entry, or generic program unit. A *formal parameter* is an identifier used to denote the named entity in the unit body. An *actual parameter* is the particular entity associated with the corresponding formal parameter in a subprogram call, entry call, or generic instantiation. A *parameter mode* specifies whether the parameter is used for input, output or input-output of data. A *positional parameter* is an actual parameter passed in positional order. A *named parameter* is an actual parameter passed by naming the corresponding formal parameter.

Pragma A pragma is an instruction to the compiler, and may be language defined or implementation defined.

Private type A private type is a type where the set of possible values is clearly defined, but not known to the users of such types. A private type is only known by the set of operations applicable to its values. A private type and its applicable operations are defined in the visible part of a module. Assignment and the predefined comparison for equality or inequality are allowed for all private types, unless the private type is marked as **restricted**.

Qualified expression A qualified expression is an expression qualified by the name of a type or subtype. It can be used to state the type or subtype of an expression, for example for an overloaded literal. It can also be used to specify a conversion to the named type when the conversion is permitted.

Range A range is a contiguous set of values of a scalar type. A range is specified by giving the lower and upper bounds for the values.

Rendezvous A rendezvous is the interaction that occurs between two parallel tasks when one task has called an entry of the other task, and a corresponding accept statement is being executed by the other task. During a rendezvous, information can be exchanged by means of parameters associated with the entry.

Representation specification Representation specifications specify the mapping between data types and features of the underlying machine that execute a program. In some cases, they completely specify the mapping, in other cases they provide criteria for choosing a mapping.

Restricted program unit A restricted program unit is a subprogram or module with a visibility restriction. The visibility restriction limits the visibility of outer units.

Restricted type A restricted type is a private type for which assignment and the predefined comparison for equality are not available.

Scalar type A scalar type is a type whose values have no components. Scalar types comprise discrete types (i.e. enumeration and integer types) and real types.

Scope The scope of a declaration is the region of text over which the declaration has an effect.

Static expression A static expression is one whose value does not depend on any dynamically computed values of variables.

Subprogram A subprogram is an executable program unit, possibly with parameters specifying its calling conventions. A *subprogram declaration* specifies the name of a subprogram and its calling conventions. A *subprogram body* specifies its execution.

Subtype A subtype of a type is obtained from the type by constraining the set of possible values of the type. The operations over a subtype are the same as those of the type from which the subtype is obtained.

Task A task is a module that may operate in parallel with other task modules. A *task family* defines a number of tasks with identical properties, each denoted by an index.

Type A type characterizes a set of values and a set of operations applicable to those values. A *type attribute* denotes an operation for the type or a property of its values. A *type definition* is a language construct introducing a type. A *type declaration* associates a name with a type introduced by a type definition.

Use clause A use clause opens the visibility to declarations given in the visible parts of given modules.

Variant A variant part specifies alternative record components in a record type. Each variant defines the components for a corresponding value of the record's discriminant.

Visibility At a given point in a program text. the declaration of an entity with a certain identifier is said to be visible if the entity is an acceptable meaning for an occurrence of the identifier. By convention an identifier is said to be visible if its declaration is visible. A *visibility restriction* is a restriction on a program unit that limits its visibility of outer units.

E. Syntax Summary

2.3

```
identifier ::=
   letter {[underscore] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter
```

2.4

```
number ::= integer_number | approximate_number

integer_number ::= integer | based_integer

integer ::= digit {[underscore] digit}

based_integer ::=
   base # extended_digit {[underscore] extended_digit}

base ::= integer

extended_digit ::= digit | letter

approximate_number ::=
      integer.integer [E exponent]
   |  integer E exponent

exponent ::= [+] integer | - integer
```

2.5

```
character_string ::= " {character} "
```

2.7

```
pragma ::=
   pragma identifier [(argument {, argument})];

argument ::= identifier | character_string | number
```

3.1

```
declaration ::=
      object_declaration          | type_declaration
   |  subtype_declaration         | private_type_declaration
   |  subprogram_declaration      | module_declaration
   |  entry_declaration           | exception_declaration
   |  renaming_declaration
```

3.2

```
object_declaration ::=
   identifier_list : [constant] type [:= expression];

identifier_list ::= identifier {, identifier}
```

3.3

```
type ::= type_definition | type_mark [constraint]

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

type_mark ::= type_name | subtype_name

constraint ::=
      range_constraint   | accuracy_constraint
   |  index_constraint   | discriminant_constraint

type_declaration ::=
   type identifier [is type_definition];

subtype_declaration ::=
   subtype identifier is type_mark [constraint];
```

3.4

```
derived_type_definition ::= new type_mark [constraint]
```

3.5

```
range_constraint ::= range range

range ::= simple_expression .. simple_expression
```

3.5.1

```
enumeration_type_definition ::=
   (enumeration_literal {, enumeration_literal})

enumeration_literal ::= identifier | character_literal
```

3.5.4

```
integer_type_definition ::= range_constraint
```

3.5.5

```
real_type_definition ::= accuracy_constraint

accuracy_constraint ::=
      digits simple_expression [range_constraint]
   |  delta simple_expression [range_constraint]
```

3.6

```
array_type_definition ::=
   array (index {, index}) of type_mark [constraint]

index ::= discrete_range | type_mark

discrete_range ::= [type_mark range] range

index_constraint ::= (discrete_range {, discrete_range})
```

3.6.2

```
aggregate ::=
   (component_association {, component_association})

component_association ::=
   [choice {| choice} => ] expression

choice ::= simple_expression | discrete_range | others
```

3.7

```
record_type_definition ::=
   record
      component_list
   end record

component_list ::=
   {object_declaration} [variant_part] | null;

variant_part ::=
   case discriminant of
      {when choice {| choice} =>
         component_list}
   end case;

discriminant ::= constant_component_name
```

3.7.3

```
discriminant_constraint ::= aggregate
```

3.8

```
access_type_definition ::= access type
```

4.1

```
name ::=
      identifier              | indexed_component
   |  selected_component | predefined_attribute

indexed_component ::= name(expression {, expression})

selected_component ::= name . identifier

predefined_attribute ::= name ' identifier
```

4.2

```
literal ::=
   number | enumeration_literal | character_string | null
```

4.3

```
variable ::= name [(discrete_range)] | name.all
```

4.4

```
expression ::=
      relation {and relation}
   |  relation {or relation}
   |  relation {xor relation}

relation ::=
      simple_expression [relational_operator simple_expression]
   |  simple_expression [not] in range
   |  simple_expression [not] in type_mark [constraint]

simple_expression ::=
   [unary_operator] term {adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary [** primary]

primary ::=
      literal | aggregate | variable | allocator
   |  subprogram_call | qualified_expression | (expression)
```

4.5

```
logical_operator        ::=  and | or | xor

relational_operator     ::=  =   | /= | < | <= | > | >=

adding_operator         ::=  +   | -  | &

unary_operator          ::=  +   | -  | not

multiplying_operator    ::=  *   | /  | mod

exponentiating_operator ::=  **
```

4.6

```
qualified_expression ::=
   type_mark(expression) | type_mark aggregate
```

4.7

```
allocator ::= new qualified_expression
```

5

```
sequence_of_statements ::= {statement}

statement ::=
      simple_statement | compound_statement
    | <<identifier>> statement

simple_statement ::=
      assignment_statement   | subprogram_call_statement
    | exit_statement         | return_statement
    | goto_statement         | assert_statement
    | initiate_statement     | delay_statement
    | raise_statement        | abort_statement
    | code_statement         | null;

compound_statement ::=
      if_statement       | case_statement
    | loop_statement     | accept_statement
    | select_statement   | block
```

5.1

```
assignment_statement ::=    variable := expression;
```

5.2

```
subprogram_call_statement ::= subprogram_call;

subprogram_call ::=
   subprogram_name
   [(parameter_association {, parameter_association})]

parameter_association ::=
      [formal_parameter :=] actual_parameter
    | [formal_parameter =:] actual_parameter
    | [formal_parameter :=:] actual_parameter

formal_parameter ::= identifier

actual_parameter ::= expression
```

5.3

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

5.4

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

condition ::=
      expression
    | expression {and then expression}
    | expression {or else expression}
```

5.5

```
case_statement ::=
   case expression of
      {when choice {| choice} => sequence_of_statements}
   end case;
```

5.6

```
loop_statement ::= [iteration_specification] basic_loop

basic_loop ::=
   loop
      sequence_of_statements
   end loop [identifier];

iteration_specification ::=
      for loop_parameter in [reverse] discrete_range
    | while condition

loop_parameter ::= identifier
```

5.7

```
exit_statement ::= exit [identifier] [when condition];
```

5.8

```
goto_statement ::= goto identifier;
```

5.9

```
assert_statement ::= assert condition;
```

6.1

```
declarative_part ::= [use_clause]
{declaration} {representation_specification} {body}

body ::= [visibility_restriction] unit_body | body_stub

unit_body ::=
   subprogram_body | module_specification | module_body
```

6.2

```
subprogram_declaration ::=
      subprogram_specification;
    | subprogram_nature designator is generic_instantiation;

subprogram_specification ::= [generic_clause]
   subprogram_nature designator [formal_part]
   [return type_mark [constraint]]

subprogram_nature ::= function | procedure

designator ::= identifier | character_string

formal_part ::=
   (parameter_declaration {; parameter_declaration})

parameter_declaration ::=
   identifier_list : mode type_mark [constraint] [:= expression]

mode ::= [in] | out | in out
```

6.4

```
subprogram_body ::=
    subprogram_specification is
      declarative_part
    begin
      sequence_of_statements
  [ exception
      {exception_handler}]
    end [designator];
```

6.7

```
block ::=
   [ declare
       declarative_part]
     begin
       sequence_of_statements
   [ exception
       {exception_handler}]
     end [identifier];
```

7.1

```
module_declaration ::=
       [visibility_restriction] module_specification
   |   module_nature identifier [(discrete_range)]
           is generic_instantiation;

module_specification ::=
   [ generic_clause]
     module_nature identifier [(discrete_range)] [is
       declarative_part
   [ private
       declarative_part]
     end [identifier]];

module_nature ::= package | task

module_body ::=
     module_nature body identifier is
       declarative_part
   [ begin
       sequence_of_statements]
   [ exception
       {exception_handler}]
     end [identifier];
```

7.4

```
private_type_declaration ::=
   [restricted] type identifier is private;
```

8.3

```
visibility_restriction ::= restricted [visibility_list]

visibility_list ::= (unit_name {, unit_name})
```

8.4

```
use_clause ::= use module_name {, module_name};
```

8.5

```
renaming_declaration ::=
       identifier  : type_mark   renames name;
   |   identifier  : exception   renames name;
   |   subprogram_nature designator renames [name.]designator;
   |   module_nature identifier renames name;
```

9.3

```
initiate_statement ::=
   initiate task_designator {, task_designator};

task_designator ::= task_name [(discrete_range)]
```

9.5

```
entry_declaration ::=
   entry identifier [(discrete_range)] [formal_part];

accept_statement ::=
   accept entry_name [formal part] [do
     sequence_of_statements
   end [identifier]];
```

9.6

```
delay_statement ::= delay simple_expression;
```

9.7

```
select_statement ::=
    select
       [when condition =>]
          select_alternative
   { or [when condition =>]
          select_alternative}
   [ else
       sequence_of_statements]
     end select;

select_alternative ::=
       accept_statement  [sequence_of_statements]
   |   delay_statement   [sequence_of_statements]
```

9.10

```
abort_statement ::= abort task_designator {,task_designator};
```

10.1

```
compilation ::= {compilation_unit}

compilation_unit ::=
  [visibility_restriction][separate] unit_body
```

10.2

```
body_stub ::=
      subprogram_specification is separate;
  |   module_nature body identifier is separate;
```

11.1

```
exception_declaration ::= identifier_list : exception;
```

11.2

```
exception_handler ::=
   when exception_choice {| exception_choice} =>
      sequence_of_statements

exception_choice ::= exception_name | others
```

11.3

```
raise_statement ::= raise [exception_name];
```

12.1

```
generic_clause ::=
   generic [(generic_parameter {; generic_parameter})]

generic_parameter ::=
      parameter_declaration
  |   subprogram_specification [is [name.]designator]
  |   [restricted] type identifier
```

12.2

```
generic_instantiation ::=
   new name [(generic_association {, generic_association})]

generic_association ::=
      parameter_association
  |   [formal_parameter is] [name.]designator
  |   [formal_parameter is] type_mark
```

13

```
representation_specification ::=
      packing_specification        | length_specification
  |   record_type_representation   | address_specification
  |   enumeration_type_representation
```

13.1

```
packing_specification ::= for type_name use packing;
```

13.2

```
length_specification ::= for name use static_expression;
```

13.3

```
enumeration_type_representation ::=
   for type_name use aggregate;
```

13.4

```
record_type_representation ::=
   for type_name use
      record [alignment_clause;]
         {component_name location;}
      end record;

location ::= at static_expression range range

alignment_clause ::= at mod static_expression
```

13.5

```
address_specification ::=
   for name use at static_expression;
```

13.8

```
code_statement ::= qualified_expression;
```

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W