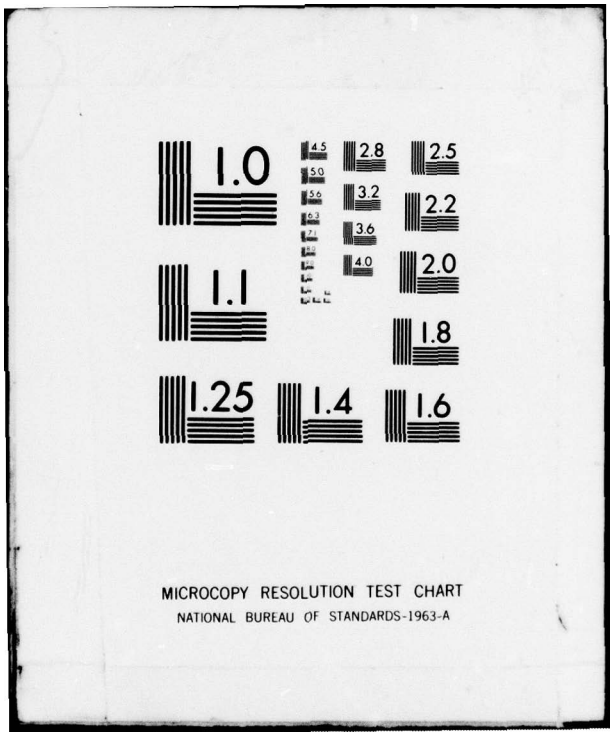
AD-A073 714 HONEYWELL SYSTEMS AND RESEARCH CENTER MINNEAPOLIS MN F/G 9/2
THE GREEN LANGUAGE. A FORMAL DEFINITION.(U)
APR 79 MDA73-77-C-0331
UNCLASSIFIED NL

1 OF 2
AD
A073714


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(9) PRELIMINARY DRAFT

(6) THE GREEN LANGUAGE
A FORMAL DEFINITION

(15) MDA73-77-C-0331

(12) 112p.

Honeywell, Inc.
Systems and Research Center
2600 Ridgway Parkway, Minneapolis, MN 55413

and

Cii Honeywell Bull
68 Route de Versailles
78430 Louvecinnes, France

LEVEL II

(11) April 1979

MDA903-77-C-0331

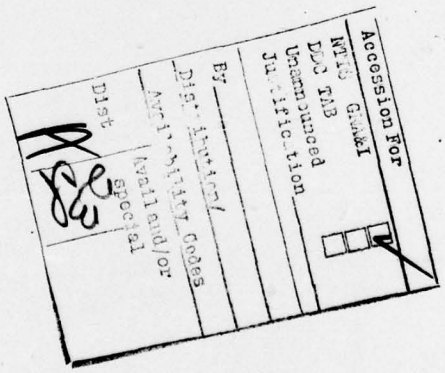
402 349

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DDC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Preface to the Preliminary Draft

This document is a Preliminary Draft of the Formal Definition of the Green Programming Language. As such, it indicates the structure and style of the final document. At this stage, the Static Semantics is well advanced but not complete, and large parts of the Dynamic Semantics are missing. Some sections (such as Chapter 4 - 6) have been worked on in detail to give a better idea of the final shape of the Formal Definition.



1. Introduction and Overview

1.1 Introduction

As a part of the DOD-1 language effort, the Steelman Report requires a formal definition (1H). This requirement is innovative and far-sighted.

Purposes

A formal definition can be put to good use:

(i) As a standard for the language, that is as a means to answer unambiguously all questions that a programmer or an implementor may raise about the meaning of a construct of the language. The formal definition should serve as a reference document for the validation of implementations and as a guideline for implementors. It should unify the user interface across implementations (e.g. error messages) and the interface between processors manipulating programs (e.g. mechanical aids for normalization and documentation of Green programs).

(ii) As a reference document for justifying the validity of optimizations and other program transformations. The only valid optimizations will be those that do not alter the meaning of a program.

(iii) As a reference document for proving properties of programs written in the language. In particular, it will allow the derivation of inference rules that can be used conveniently when proving properties of programs.

(iv) As an input for a compiler-generator when the technology becomes available. The Formal Definition given in this report is specified with enough precision to be processed, except for some straightforward notational transformations, by the experimental system SIS [Mosses].

Furthermore, the concurrent development of Green and its formal definition has already resulted in further major benefits:

- Difficulties in early drafts of the Reference Manual (such as lack of clarity, ambiguities, omissions or inconsistencies) were uncovered very early.

- Feedback was established to strive for economy of concepts in the Green language.

Requirements

When designing the formal definition of a language like Green, there are two major requirements to satisfy:

(i) The definition should be complete. If the definition is not complete its usefulness as a reference will be seriously diminished. This completeness can only be achieved by using a mathematically well-founded definitional method.

As of the Spring of 1979, however, the State of the Art in formal semantics does not allow us to offer a mathematically meaningful semantics for parallelism. This is a very serious gap in our theoretical understanding of programs. No attempt has been made here to give a dynamic semantics for task synchronization in Green, while it is hoped that all other aspects of the language are satisfactorily covered.

In all matters relating to concurrency, the readers will have to do with the textual description of the dynamic semantics that is provided, pending a scientific breakthrough.

(ii). The Formal Definition of Green is meant to be used in an industrial environment. It must not remain an academic exercise. One difficulty when trying to reach a wider audience is the notational barrier.

A great deal of effort should be spent on the style of the definition and its intuitive content, to make it accessible to the intended readership: implementors of compilers, standardization committees, educated Green programmers. Naturally, such an attempt should preserve the mathematical rigor of the definition, and should be seen merely as the development of a convenient notation.

The formal definition given here is akin to a large program. Special attention has been given to several key issues:

- The structure of the description reflects the underlying semantic concepts of the language.
- The choice of identifiers stays as close as possible to the terminology of the Reference Manual.
- The style of the description is homogeneous and uniform conventions are used throughout.

Method

There are three widely accepted methods of formally defining the semantics of a programming language.

(a) Operational Semantics

In this method, best exemplified by the Vienna Definition Method, the semantics is modelled by the behavior of an abstract machine. This has a practical appeal but also presents several problems:

(i) The mechanism of the abstract machine tends to overspecify the language since all details of machine-state transitions must be given.

(ii) It is not immediately obvious that the language has been well defined. One must rely on a proof that any execution terminates with a unique answer.

(iii) The theory of operational semantics is, in fact, rather difficult and not well-understood. Using an operational semantics to validate optimizations or to prove properties of programs is intricate because we are not well-equipped to reason logically about the behavior of a complex machine.

(b) Axiomatic Definition

This method is very popular because it is directed towards proving properties of programs. Its deficiencies, however, render it totally unsuitable for the definition of a language like Green:

(i) First, giving some properties of language constructs cannot constitute a definition, unless some proof of completeness can be given.

(ii) An axiomatic definition is not adapted to a use by implementors since many details about the dynamic semantics cannot be formalized adequately.

(iii) No complete axiomatic definition of a large programming language has ever been carried out successfully, to date. Treatment of exceptions, for example, does not fit well in this formalism. It is even doubtful that this method can accomodate it at all.

(c) Denotational Semantics

In this document, we are going to present a formal definition of Green using denotational semantics. There are several reasons for choosing this method:

(i) It allows the definition of the language to any desired level of detail.

(ii) The method has been used (with success) on a number of languages with characteristics similar to those of Green: Pascal, Algol 60, CLU, etc.

(iii) The mathematics underlying this method have been extensively investigated. The method is based on very strong theoretical foundations.

(iv) It is well-suited to proving the validity of program transformations and proving properties of programs.

A potential objection to the use of this method is the arcane style of presentation traditionally favored by its advocates. In this report, we hope to have overcome this difficulty.

Summary of Denotational Semantics

In denotational semantics, one wishes to associate to every program an abstract mathematical object called its meaning. Usually, the meaning of a program is some functional object, say a function from inputs to outputs. The mapping that specifies how one associates a meaning to every program in Green is called the denotational semantics of Green. To properly define the denotational semantics

of a language, one must first define a semantic universe, where meanings are to be found. Then one describes how to associate a meaning to every atomic component of a program and, for every construct of the language, how to derive the meaning of a compound fragment of program from the meaning of its subparts. Hence, denotational semantics is nothing but a rather large, recursive definition of a function from syntactic objects - programs - to semantic objects - input-output functions.

Defining the semantics of a language in this way naturally leads to assigning a meaning not only to complete programs but also to program fragments, a very useful mathematical property known as referential transparency. The recursive structure of the syntactic objects is well captured by the abstract syntax of Green. Section 1.2 is devoted to a detailed presentation of the abstract syntax of Green, that is of the tree form of Green programs.

There is a wide body of literature discussing the mathematical nature of the semantic domains that need to be used. At first, it is not necessary to understand in depth the mathematical theory of these domains in order to follow the semantic description of Green. Denotational semantics uses a very small number of concepts. We shall describe, in general terms, three keys ideas that pervade the whole definition.

Green is an imperative language. Understanding it requires some notion of a store. Programs use the store and update it as they are executed. Now if we wish to describe the store as abstractly as possible, that is without assuming any particular implementation, all we need to know is that it defines a mapping

STORE: LOCATIONS ---> VALUES

If s is a store and l is a location the expression s(l) will then denote the value stored at location l. To update the store, we will assume the existence of a function UPDATE that, given a store s, a location l and a value v returns a new store s' = UPDATE(s,l,v) that differs from s only by the fact that s'(l) = v. Typically, it is the purpose of an assignment statement to modify the store.

Another feature of Green is its block structure. This feature allows a given identifier to refer to different objects, depending on where it occurs in a program. To model this phenomenon abstractly, we will assume the existence of a mapping:

ENVIRONMENT: IDENTIFIERS ---> DENOTATIONS

Here again, by merely saying that an environment is such a mapping, we want to avoid describing any particular implementation of this concept. The primary purpose of declarations is to modify the environment. In Green, however, there are many other ways to alter the environment.

As a third example, let us consider the problem of describing the control mechanism of Green. At first it would not seem too easy to describe it in a referentially transparent manner. If the meaning of an assignment is some transformation of the store, the meaning of a sequence of assignments should be the composition of these transformations. But what if we wish to give meaning to a **goto** statement or an **exit** statement? How can we describe the raising of an exception, either explicitly or during the evaluation of an expression? A very general technique allows us to deal with this kind of problem in denotational

semantics. Intuitively, the idea here is to give to the semantic functions an extra parameter that specifies "what-to-do-next". This parameter is called a continuation. The meaning of a program fragment is in general also a continuation. Typically, the meaning of an assignment statement with continuation c is obtained by prefixing c with a store to store transformation. In fact, Green has a sophisticated exception mechanism, implying the use of a whole exception environment associating a continuation to each exception handler.

Continuations are not very easy to understand at first. The Static Semantics does not use any continuations, so that it is possible to become thoroughly familiar with the Formal Definition's approach before having to tackle this concept.

Style of the Definition

Given that the first objective of the Formal Definition is to serve as a reference document for implementors, a great deal of attention must be given to the choice of the meta-language, i.e. the language in which Green is to be formally described. The typographical conventions of the Oxford School are not suitable for such an audience, due to an intensive use of Greek letters and diacritical signs. The notation proposed by Peter Mosses, (which is used as input for his system SIS) is a much better candidate. Mosses' notation is elegant, machine readable, convenient to use for anybody familiar with applicative programming and efficient in its treatment of abstract syntax. In this report, we have tried to go even further towards usual programming convention in using an (applicative) subset of Green itself as a meta-language.

A minor notational extension was needed in order to allow procedures as arguments and results. Italics are used to avoid confusion between language and metalanguage. Identifiers in distinct fonts are considered to be distinct. It is hoped that the increased understandability of the Formal Definition will compensate for the loss of elegance.

In keeping with the goal of minimizing the number of new notations, we have attempted to stay close to the terminology of the Reference Manual, refraining from introducing new names unless they were absolutely necessary. Furthermore, rather than presenting the Formal Definition as a completely separate document, we have followed the structure of the Reference Manual. The equations of the Formal Definition intend to make more explicit the English text in the Reference Manual. Experience with the Formal Definition will show whether this is the right approach.

As a final remark, the reader will notice that we make use of the abstraction facility of Green. It may seem unfortunate that we could not avoid using one of the seemingly more advanced features of the language. But in fact, all we really need is a way to specify a collection of related functions together with their types. This concept is very familiar in mathematics as an algebra. Similarly, the use of the generic facility corresponds directly to the notion of a polymorphic function (or functional) in mathematics. In fact, all (value returning) procedures defined in the document are functions in the mathematical sense. The sublanguage of Green that is used is purely applicative and the only "side effects" are the construction of new objects.

1.2 Abstract Representation of Programs

In this section, we present a standard way of representing programs. It is to be used not only to define the semantics of the Green language but also as a standard interface between all processors manipulating Green programs. Programs are represented as trees, called Abstract Syntax Trees. These trees are defined with the help of the Green's encapsulation facility, so as not to preclude subsequent efficient implementation.

1.2.1 Motivations

Since the meaning of programs will be defined recursively on their structure, it is necessary to specify with great precision what this structure is before developing the Formal Definition per se. On the other hand, quite apart from the Formal Definition, there is considerable interest in standardizing the representation of programs. This standard representation will play a crucial part in the harmonious development of the programming support, a collection of issues addressed in Pebbleman. Typical tools that are to benefit from such a definition are: syntax-oriented editors, interpreters and compilers, documentation and normalization aids, program analyzers, optimizers, verification tools.

1.2.2 Requirements

We now list some requirements that an abstract representation must satisfy to be effective as a standard:

(a) It must be possible to implement it efficiently on a variety of machines.

(b) It must reflect the structure of programs. For example, it must be easy to recognize and isolate program fragments such as statements, procedures, declarations, expressions, identifiers, etc...

(c) It must be easy to manipulate and modify.

(d) It must include all meaningful information contained in the original program text. In particular it must be possible to restore the program text from the representation, up to minor standardizations.

(e) It should not be cluttered with irrelevant information.

(f) It must have a simple and usable mathematical definition since it will be a foundation for the Formal Definition.

(g) Finally, as a matter of course, it must allow the representation of any legal Green program.

Requirements (b) and (c) rule out the textual representation of programs. It is easy to see that many processors would need a "parser" as a mandatory front end. It would also be a mistake to use a parse tree as usually produced by a parser: such trees depend on the parsing method used and are cluttered with irrelevant details (Requirement (e)).

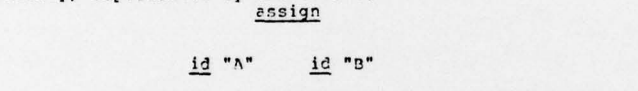
Common intermediate languages designed for optimization fail requirements (b) through (d). Using abstract syntax, a method put forward in the early sixties [McCarthy, Landin] is very natural, simple and meets requirements (a) through (g).

1.2.3 Abstract Syntax Trees

The essential idea underlying abstract syntax is the treatment programs and program fragments as trees. For example, the assignment

A := B

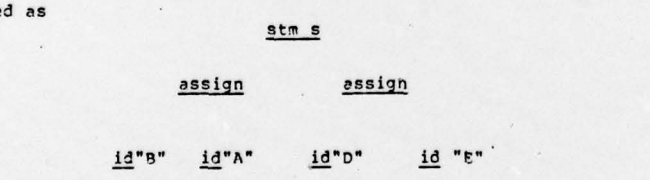
will be (pictorially) represented by the tree t.



Each node in the tree is labeled by a construct. In our notation, the construct labeling the top node of the tree t is denoted by KIND(t). Here KIND(t) = assign. The subtree representing the left-hand-side of the assignment is denoted by SON(1,t) and the subtree denoting the right-hand-side by SON(2,t). The whole Green language is defined using 126 constructs. Most constructs label trees with a fixed number of sons. These constructs are said to be of fixed arity. To represent lists, it is necessary to use nodes that may have an arbitrary number of sons. For example the fragment

```
B := A;
D := E;
```

is represented as



The construct assign is binary while stm s is a list construct. The node labeled stm s could have an arbitrary number of sons.

Notations: All Green constructs are written in italics. List constructs have names ending in _s, like stm_s, exp_s, or decl_s.

Not all trees labeled with constructs are abstract syntax trees. A grammar imposes a restriction on the strings of terminal symbols that are sentences of the language it defines. The Green abstract syntax is similarly defined by a tree grammar. It specifies precisely which trees are Green trees. Let us define a sort to be a set of constructs. The abstract syntax of Green is specified with the help of 57 sorts. If the root of a tree t is a construct belonging to sort s, we say that t is of sort s. The entire abstract syntax of Green is completely specified by giving, for each construct, its arity as well as the sort of each son.

Note that list constructs are homogeneous: all constituents of a list must be of the same sort.

Notations.

(a) Sorts are written in italics and is capitalized (e.g. COND). When a sort is a singleton sort (i.e. it contains a single construct), it has the same name as its member, but capitalized. Furthermore, since list constructs are characterized by the common sort of their constituents, their name always reflects that sort. As an example, a node labeled stm_s has subtrees of sort STM, a node labeled decl_s has subtrees of sort DECL.

(b) A notation similar to BNF has been used to specify the sorts. When writing for example:

    COND ::= EXP | condition

we mean that COND is the union of sort EXP and the singleton set {condition}. Since sorts and constructs are distinguished typographically, the symbol | is used without ambiguity. For each construct, a sequence of sorts is given. For example the specification

    if -> CONDITIONAL_S STM_S

means that the first son of an if construct is of sort CONDITIONAL_S and the second son is of sort STM_S.

Formally,

    SORT_OF_SON(if,1) = CONDITIONAL_S
    SORT_OF_SON(if,2) = STM_S

In the case of list constructs, the fact that all constituents belong to the same set is emphasized by the use of three dots as in

    stm_s -> STM ...

The complete Abstract Syntax of Green is given in Appendix A.

1.2.4 Encapsulation of the Abstract Syntax

To be certain that the abstract syntax of Green can be used as a standard for the representation of Green programs, we could define it as a Green data structure. This would not however leave enough room for efficient implementation and would involve unnecessary and harmful overspecification. Rather, we specify here the visible part of Green instead, we specify here the visible part of Green packages that provide the abstract syntax of Green and the tools for the manipulation of Green trees. Notice that the procedure MAKE is overloaded. This overloading will be resolved on the basis of the number of arguments handed to it in any call. The procedure MAKE must be programmed using the KIND and SORT_OF_SON procedures provided in the package GREEN_SYNTAX, to check that it is not asked to build unlawful Green trees. Similarly, the constructor procedure EMPTY checks that its argument is a construct of arbitrary arity.

Most processors will find the selector function SON perfectly adequate. For the Formal Definition, where readability is of prime importance, we have assumed the existence of a third package, GREEN_SELECTORS. This package allows us to refer to subtrees by name rather than by position. A simple convention for the names of the selectors has been followed in the Formal Definition: for each sort a selector function is defined that is named after the sort. Assume now, for example, that "if" is a tree with a root labeled if. Instead of writing:

```
SON(1,if)
```

we may write

```
CONDITION_S(if)
```

In cases like the binary construct pair that has more than one son of the same sort, numbering is used. Thus EXP1(pair) and EXP2(pair) return the first and second son of the tree pair, respectively, as both are of sort EXP.

1.3 Structure and Notations

In the informal description of the semantics of Green given in the Reference Manual, one can distinguish three kinds of concerns:

(i) Some features of the language are provided to shorten the text of programs or to increase their readability. These features are best explained as combinations of other possibilities of Green.

(ii) A group of specifications are intended to delineate the class of legal programs, within the class of syntactically correct ones. Considerations such as the need to declare every identifier used, coherence in the use of types and resolution of ambiguity in the use of overloading, are in this category.

(iii) The rest of the informal definition concerns the behavior of programs during execution.

The Formal Definition is structured to reflect these quite distinct concerns.

1.3.1 Normalization

This part of the Formal Definition specifies transformations of the abstract syntax tree that do not require any type information. These transformations are performed to eliminate the use of some notational conveniences or to check simple syntactic constraints. They are defined by functions mapping TREE's to TREE's and given in Appendix B. Whenever these functions are sufficiently simple (i.e. involve no context), the text includes their description as a rewriting rule.

Example:

```
[ if CONDITIONAL_S else STM_S end if; ] ->
[ if CONDITIONAL_S elsif true then STM_S end if; ]
```

The kind of constraints dealt with by normalizations must require only little contextual information, i.e. no information about types. For example, the Manual states:

> "Within the sequence of statements of a subprogram or module body, different labels must have different identifiers."

This check is supposed to be performed in this normalization phase.

1.3.2 Static Semantics

This part of the Formal Definition is concerned with what is usually called type_checking. A type checker is given as a mapping from abstract syntax trees to an extended abstract syntax tree. This is intended to mimic the concepts of "compile time" checks as opposed to "run time" checks. Type_checked programs contain all type information needed at run time. In this way dynamic semantics will not need to carry a static environment.

The Static Semantics of Green has to deal with the following:

1. It must check that the declarations are valid, i.e. there is no repeated declaration of the same designator in the same scope. It must check that all designators are declared.

2. It must check that all designators are used in a manner that is consistent with their type.

3. It must carry out the evaluation of static expressions where needed.

4. All information on types of designators must be used to generate an extended abstract syntax tree. This includes:

4.1 Detecting and eliminating all overloading

4.2 Reordering actual parameters in subprogram calls. Once it has been processed, a subprogram call will list all its parameters in named parameter associations.

4.3 Normalizing aggregates as lists of named component associations.

4.4 Resolving ambiguities between indexed component, qualified expression and subprogram call.

5. Exception names are made unique within a program

6. The dot notation is systematically used to access identifiers visible through a use list.

The Static Semantics presented here is parameterized by

- a "machine" that abstracts away the structure of the static environment, where information regarding the type of designators is recorded. The external behavior of this "machine" is defined by a set of functions that
  - build or select type denotations
  - declare or access designators
- a "machine" which abstracts auxiliary functions used
  - to solve overloading
  - check for side effects of functions and value returning procedures

1.3.3 Dynamic Semantics

The Static Semantics is described as a transformation performed on abstract syntax trees. The Dynamic Semantics corresponds more to the customary notion of interpretation. The meaning of each construct is defined recursively on type checked abstract syntax trees. Information about the identifiers in the program (e.g. the value of a constant, the constraints associated with a subtype) is recorded in the dynamic environment.

The functions used in Dynamic Semantics are partitioned into three groups, following to the terminology of the Reference Manual:

(a) Those defining the elaboration of declarations (Prefix ELAB).

(b) Those defining the evaluation of expressions (Prefix EVAL).

(c) Those defining the execution of statements (Prefix EXEC).

The dynamic semantics is also parameterized by

- an abstract machine that provides a model of storage allocation
- a set of definitions which characterize the restrictions of a concrete machine (minimum and maximum value for integers, etc).

1.3.4 Treatment of Errors

Errors discovered during Normalization and during the checking of the Static Semantics are reported by inserting a special construct in the Abstract Syntax Tree at the lowest meaningful level. The Dynamic Semantics is only defined on trees which do not contain such errors. In this way, the place and reason for an error are defined. Error messages can be standardized accordingly.

Errors occurring during the execution of a program raise the appropriate exceptions, as prescribed by the semantics of Green.

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.

Syntax:

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

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter
```

Abstract Syntax:

```
id -> -- [lexical unit ]

ID ::= id
```

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

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.

Syntax:

```
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
```

Abstract Syntax:

```
int number  -> -- [ lexical unit ]
real number -> -- [ lexical unit ]
```

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.

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).

Syntax:

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

Abstract Syntax:

```
string ->  -- [ lexical unit ]
```

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.

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.

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.

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.

Syntax:

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

Abstract Syntax:

```
decl          -> NATURE   DESIGNATOR_S   DESCRIPTION
designator_s  -> DESIGNATOR ...        -- [DESIGNATOR, ... ]

NATURE ::= constant | variable | type      | subtype | private | procedure | function |
           package  | entry    | exception | in      | out     | in out
DESIGNATOR_S ::= designator_s
DESIGNATOR   ::= id | string
DESCRIPTION  ::= instantiation | object | renaming | type | unit | void

-- Depending on its nature, a decl may appear as:
-- [DESIGNATOR_S: constant DESCRIPTION]
-- [DESIGNATOR_S: DESCRIPTION]
-- [type            DESIGNATOR_S is DESCRIPTION]
-- [subtype         DESIGNATOR_S is DESCRIPTION]
-- [restricted type DESIGNATOR_S is DESCRIPTION]
-- [procedure       DESIGNATOR_S    DESCRIPTION]
-- [function        DESIGNATOR_S    DESCRIPTION]
-- [package         DESIGNATOR_S    DESCRIPTION]
-- [task            DESIGNATOR_S    DESCRIPTION]
-- [entry           DESIGNATOR_S    DESCRIPTION]
-- [DESIGNATOR_S: exception]
-- [DESIGNATOR_S: in DESCRIPTION]
-- [DESIGNATOR_S: out DESCRIPTION]
-- [DESIGNATOR_S: in out DESCRIPTION]
```

Description:

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.

Static Semantics:

```
procedure CHECK_DECL_S(decl_s: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
begin
   if IS_EMPTY(decl_s) then
      return TREE_ENV(EMPTY(decl_s), local);
   else
      declare
      tree_env : constant TREE_ENV := CHECK_DECL(HEAD(decl_s), env, local);
      local2   : constant S_ENV    := ENV(tree_env);
      env2     : constant S_ENV    := NESTED_ENV(env, local2);
      tree_env2: constant TREE_ENV := CHECK_DECL_S(TAIL(decl_s), env2, local2);
      begin
         return TREE_ENV(PRE(TREE(tree_env), TREE(tree_env2)), ENV(tree_env2));
      end;
   end if;
end CHECK_DECL_S;

procedure CHECK_DECL(decl: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
begin
   case KIND(DESCRIPTION(decl)) of
      when instantiation => DECL_INSTANTIATION(decl, env, local);
      when object        => DECL_OBJECT       (decl, env, local);
      when renaming      => DECL_RENAMING     (decl, env, local);
      when type          => DECL_TYPE         (decl, env, local);
      when unit          => DECL_UNIT         (decl, env, local);
      when void          => DECL_EXCEPTION    (decl, env, local);
   end case;
end CHECK_DECL;

procedure CHECK_DESIGNATOR_S(designator_s: TREE; env: S_ENV; den: S_DEN) return TREE_ENV is
begin
   if IS_EMPTY(designator_s) then
      return TREE_ENV(designator_s, env);
   else
      declare
         tree_env : constant TREE_ENV := CHECK_DESIGNATOR  (HEAD(designator_s), env, den);
         tree_env2: constant TREE_ENV := CHECK_DESIGNATOR_S(TAIL(designator_s), ENV(tree_env), den);
      begin
         return TREE_ENV(PRE(TREE(tree_env), TREE(tree_env2)), ENV(tree_env2));
      end;
   end if;
end CHECK_DESIGNATOR_S;
```

```
procedure CHECK_DESIGNATOR(designator: TREE; env: S_ENV; den: S_DEN) return TREE_ENV is
begin
   if IS_OVERLOADABLE(DEN_OF(designator, env), den) then
      return TREE_ENV(designator, OVERLOAD(designator, env, den));
   elsif IS_FORWARD(DEN_OF(designator, env)) then
      return TREE_ENV(designator, UPDATE(designator, env, den));
   elsif not(IS_DECLARED(designator, env)) then
      return TREE_ENV(designator, UPDATE(designator, env, den));
   else
      return TREE_ENV(ALREADY_DECLARED, env);
   end if;
end CHECK_DESIGNATOR;
```

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.

Syntax:

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

identifier_s ::= identifier {, identifier}
```

Abstract Syntax:

```
object -> TYPE   EXP_VOID    -- [TYPE := EXP] or [TYPE]
```

Description:

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).

Static Semantics:

```
procedure DECL_OBJECT(decl: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
   nature      : constant TREE := NATURE      (decl);
   designator_s: constant TREE := DESIGNATOR_S(decl);
   object      : constant TREE := DESCRIPTION (decl);
   begin
      case KIND(nature) of
         when constant | -- [DESIGNATOR_S: constant DESCRIPTION]
```

```
            variable | -- [DESIGNATOR_S: DESCRIPTION]
            in       | -- [DESIGNATOR_S: in DESCRIPTION]
            out      | -- [DESIGNATOR_S: out DESCRIPTION]
            in out   | -- [DESIGNATOR_S: in_out DESCRIPTION]
            =>
          declare
             tree_des_den : constant TREE_DES_DEN := CHECK_OBJECT(object, env);
             object2      : constant TREE        := TREE(tree_des_den);
             den          : constant S_DEN       := DEN(nature, DESDEN(tree_des_den));
             tree_env     : constant TREE_ENV    := CHECK_DESIGNATOR_S(designator_s, local, den);
             designator_s2: constant TREE        := TREE(tree_env);
          begin
             return TREE_ENV(MAKE(decl, nature, designator_s2, object2), ENV(tree_env));
          end;
       when others =>
          return TREE_ENV(SYNTAX_ERROR, local);
    end case;
end DECL_OBJECT;


procedure CHECK_OBJECT(object: TREE; env: S_ENV) return TREE_DES_DEN) is
   tree_type_den: constant TREE_TYPE := CHECK_TYPE_CONSTRAINT(TYPE(object), env);
   exp_void     : constant EXP_VOID  := CHECK_EXP_VOID(EXP_VOID(object), env);
   des_den      : constant S_DES_DEN := DES_DEN(TYPE_DEN(tree_type_den), exp_void);
begin
   return TREE_DES_DEN(MAKE(object, TREE(tree_type), exp_void), des_den);
end CHECK_OBJECT;
```

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.

Syntax:

```
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];
```

Abstract Syntax:

```
decl        -> NATURE  DESIGNATOR_S   DESCRIPTION
constrained -> NAME    CONSTRAINT

TYPE ::= access  | array        | comp_s             | constrained |
         derived | designator_s | fixed              | float       |
         integer | private      | restricted private | void

CONSTRAINT ::= fixed      | float   | comp_assoc_s | pair |
               typed_pair | range_s | void
```

Description:

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.

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.

Syntax:

```
range_constraint ::= range range

range ::= simple_expression .. simple_expression
```

Abstract Syntax:

```
pair       -> EXP    EXP      -- [EXP .. EXP]
typed pair -> NAME   PAIR     -- [NAME range PAIR]

RANGE ::= pair | typed pair
PAIR  ::= pair
```

Description:

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.

Syntax:

```
enumeration_type_definition ::=
   (enumeration_literal {, enumeration_literal})

enumeration_literal ::= identifier | character_literal
```

Abstract Syntax:

```
designator_s -> DESIGNATOR ...     [DESIGNATOR, ...]

DESIGNATOR   ::= string | id
```

Description:

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 values 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).

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.

Syntax:

integer_type_definition ::= range_constraint

Abstract Syntax:

integer -> RANGE

Description:

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.

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.

Syntax:

```
real_type_definition ::= accuracy_constraint

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

Abstract Syntax:

```
float -> EXP   RANGE_VOID     -- [digits EXP RANGE_VOID]
fixed -> EXP   RANGE_VOID     -- [delta  EXP RANGE_VOID]

RANGE_VOID ::= void | RANGE
```

Description:

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.

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.

Syntax:

```
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})
```

Abstract Syntax:

```
array    -> BOUNDS S   TYPE      -- [array BOUNDS S of TYPE]
bounds s -> BOUNDS ...           -- [BOUNDS , ...]

BOUNDS   ::= id       | indexed | pair | predefined |
             selected | typed pair
BOUNDS S ::= bounds s
```

Description:

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

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.

3.6.2 Aggregates

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

Syntax:

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

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

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

Abstract Syntax:

```
comp_assoc_s -> COMP_ASSOC ...        -- [(COMP_ASSOC, ...)]
named        -> CHOICE_S   EXP        -- [CHOICE_S => EXP]
choice_s     -> CHOICE ...            -- [CHOICE | ... ]

COMP_ASSOC ::= named        | all      | allocator       | binary    | call |
               comp_assoc_s | id       | int_number      | indexed   |
               membership   | null     | predefined      | qualified |
               real_number  | selected | selected_string | slice     |
               string       | unary

CHOICE_S ::= choice_s
CHOICE   ::= EXP | others | RANGE
```

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.

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;
```

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.

Syntax:

```
record_type_definition ::=
   record
      component_s
   end record

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

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

discriminant ::= constant component name
```

Abstract Syntax:

```
comp_s       -> COMP ...                -- [record COMP ... end record]
variant_part -> NAME   VARIANT_S        -- [case NAME of VARIANT_S]
variant_s    -> VARIANT ...             -- [VARIANT ...]
variant      -> CHOICE_S   COMP_S       -- [when CHOICE_S => COMP_S]

COMP      ::= decl | null | variant_part
VARIANT_S ::= variant_s
VARIANT   ::= variant
```

Description:

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).

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.

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**.

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.

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.

Syntax:

    access_type_definition ::= **access** type

Abstract Syntax:

```
access -> TYPE      -- [access TYPE]
```

Description:

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).

4. Names, Variables, and Expressions

4.1 Names

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

Syntax:

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

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

selected_component   ::= name . identifier

predefined_attribute ::= name ' identifier
```

Abstract Syntax:

```
id          ->                    -- [ lexical unit]
indexed     -> NAME EXP_S         -- [ NAME(EXP_S) ]
predefined  -> NAME ID            -- [ NAME'ID ]
selected    -> NAME ID            -- [ NAME.ID ]

NAME ::= all | id | indexed | predefined | selected | slice
```

Description:

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

Static Semantics:

```
CHECK_NAME(name: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
begin
   case KIND(name) of
      when all        => CHECK_ALL       (name, type_den, env);
      when id         => CHECK_ID        (name, type_den, env);
      when indexed    => CHECK_INDEXED   (name, type_den, env);
      when predefined => CHECK_PREDEFINED(name, type_den, env);
      when selected   => CHECK_SELECTED  (name, type_den, env);
      when slice      => CHECK_SLICE     (name, type_den, env);
```

```
      end case;
   end CHECK_NAME;

   function CHECK_ID(id: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   begin
      if IS_COMPATIBLE(type_den, TYPE_DEN_OF(id, env)) then
         return id;
      else
         return ID_INCOMPATIBLE_WITH_TYPE;
      end if;
   end CHECK_ID;
```

4.1.1 Indexed Components

Description:

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.

Static Semantics:

```
   function CHECK_INDEXED(indexed: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
      den      : constant S_DEN      := DEN_OF  (NAME(indexed), env);
      name_type: constant S_TYPE_DEN := TYPE_DEN(den);
   begin
      if IS_ARRAY(name_type) and IS_COMPATIBLE(type_den, COMPONENT_TYPE(name_type))
         then return MAKE(indexed, NAME(indexed), CHECK_INDEX_S(EXP_S(indexed), type_den, env));
      elsif IS_ACCESS(name_type) then
         declare
            name_type1: constant S_TYPE_DEN := OBJECT_TYPE(name_type);
         begin
            if IS_ARRAY(name_type) and IS_COMPATIBLE(type_den, COMPONENT_TYPE(name_type1))
               then return MAKE(indexed, NAME(indexed), CHECK_INDEX_S(EXP_S(indexed), type_den, env)
```

```
            else return INDEX_TYPE_INCOMPATIBLE;;
            end if;
         end;
      elsif IS_TASK(name_type) then
         if IS_ALONE(EXP_S(indexed)) then
            return CHECK_TASK(indexed, type_den, env);
         else return ONLY_ONE_EXP_ALLOWED;
         end if;
      elsif IS_ENTRY(name_type) then
         if IS_ALONE(EXP_S(indexed)) then
            return CHECK_ENTRY_FAMILY(indexed, type_den, env);
         else return ONLY_ONE_INDEX_ALLOWED;
         end if;
      else return TYPE_INDEX_NAME_INCOMPATIBLE;
      end if;
   end CHECK_INDEXED;
```

4.1.2 Selected Components

Description:

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.

Static Semantics:

```
function CHECK_INDEX_S(exp_s: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
begin
   if IS_EMPTY(exp_s) then return void;
   else if IS_ARRAY(type_den) then
      return PRE(CHECK_EXP(HEAD(exp_s), INDEX_TYPE(type_den), env),
```

```
                    CHECK_INDEX_S(TAIL(exp_s), COMPONENT_TYPE(type_den), env));
      elsif IS_VOID(type_den) then
         return INCOMPATIBLE_TYPE;
      else
         return PRE(CHECK_EXP(HEAD(exp_s), type_den, env);
      end if;
   end CHECK_INDEX_S;
```

4.1.3 Predefined Attributes

Description:

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.

Static Semantics:

4.2 Literals

A literal denotes an explicit value of a given type.

Syntax:

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

Abstract Syntax:

```
int number  ->  -- [ lexical unit ]
real number ->  -- [ lexical unit ]
id          ->  -- [ lexical unit ]
string      ->  -- [ lexical unit ]
null        ->  -- [ null ]
```

Informal Semantics:

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.

Static Semantics:

```
procedure CHECK_LITERAL(literal: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
begin
   if IS_VALID_LITERAL(literal, type_den, env) then
      return literal;
   else
      return LITERAL_INCOMPATIBLE_WITH_TYPE;
   end if;
end CHECK_LITERAL;

procedure IS_VALID_LITERAL(literal: TREE; type_den: S_TYPE_DEN; env: S_ENV) return BOOLEAN is
begin
   case KIND(literal) of
      when int_number  => return IS_INTEGER(type_den);
      when real_number => return IS_REAL(type_den);
      when id          => return IS_VALID_ENUMERATION(literal, type_den, env);
      when string      => return IS_STRING(type_den) or IS_CHARACTER(literal) and
                           IS_VALID_ENUMERATION(literal, type_den, env);
      when null        => return IS_ACCESS(type_den);
   end case;
end IS_VALID_LITERAL;

procedure IS_VALID_ENUMERATION(literal: TREE; type_den: S_TYPE_DEN; env: S_ENV) return BOOLEAN is
begin
   return IS_COMPATIBLE(type_den, TYPE_OF(literal, env));
end IS_VALID_ENUMERATION;
```

Dynamic Semantics:

```
procedure EVAL_LITERAL(literal: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
begin
   return cont(VAL (literal));
end EVAL_LITERAL;

procedure VAL(literal: TREE) return VAL is
begin
   -- maps literal to an abstract value
end VAL;
```

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.

Syntax:

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

Abstract Syntax:

```
all   -> NAME          [NAME.all]
slice -> NAME  RANGE [NAME    (RANGE)]

NAME ::= all | id | indexed | predefined | selected | slice
```

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.

4.4 Expressions

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

Syntax:

```
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)
```

Abstract Syntax:

```
exp s      -> EXP ...  -- [ EXP , ... ]

EXP        ::= binary     | unary       | membership
             | int number | real number | string | null | aggregate
             | NAME       | allocator   | call   | qualified
EXP LIST   ::= exp s
TYPE RANGE ::= constrained | pair | typed pair
```

Description:

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.

Static Semantics:

```
procedure CHECK_EXP(exp: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   exp2: constant TREE := NORMALIZE_EXP(exp, type_den, env);
begin
   if KIND(exp2) = id and not IS_ENUMERATION(type_den) then
      return CHECK_NAME    (exp2, type_den, env);
   elsif IS_LITERAL(exp2) then
      return CHECK_LITERAL(exp2, type_den, env);
   elsif IS_NAME(exp2) then
      return CHECK_NAME    (exp2, type_den, env);
   else
      case KIND(exp2) of
         when binary       => return CHECK_BINARY    (exp2, type_den, env);
         when unary        => return CHECK_UNARY     (exp2, type_den, env);
         when membership   => return CHECK_MEMBERSHIP(exp2, type_den, env);
         when comp_assoc_s -> return CHECK_AGGREGATE (exp2, type_den, env);
         when allocator    => return CHECK_ALLOCATOR (exp2, type_den, env);
         when call         => return CHECK_CALL      (exp2, type_den, env);
         when qualified    -> return CHECK_QUALIFIED (exp2, type_den, env);
      end case;
   end if;
end CHECK_EXP;


procedure NORMALIZE_EXP(exp: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
begin  -- discovers one-component aggregates
   if IS_ARRAY(type_den) and IS_VALID_ARRAY_COMP(exp, type_den, env)
      return PRE(MAKE(comp_assoc, exp), EMPTY);
   elsif IS_RECORD(type_den) and IS_VALID_RECORD_COMP(exp, type_den, env)
      return PRE(MAKE(comp_assoc, exp), EMPTY);
   else
      return exp;
   end if;
end NORMALIZE_EXP;


procedure IS_VALID_EXP(exp: TREE; type_den: S_TYPE_DEN; env: S_ENV) return BOOLEAN is
   exp2: constant TREE := NORMALIZE_EXP(exp, type_den, env);
begin
   if KIND(exp2) = id and not IS_ENUMERATION(type_den) then
      return IS_VALID_NAME    (exp2, type_den, env);
   elsif IS_LITERAL(exp) then
      return IS_VALID_LITERAL(exp2, type_den, env);
   elsif IS_NAME(exp2) then
      return IS_VALID_NAME    (exp2, type_den, env);
   else
      case KIND(exp2) of
         when binary     => return IS_VALID_BINARY    (exp2, type_den, env);
```

```
            when unary      => return IS_VALID_UNARY    (exp2, type_den, env);
            when membership => return IS_BOOLEAN        (type_den);
            when aggregate  => return IS_VALID_AGGREGATE(exp2,type_den,env);
            when allocator  => return IS_VALID_ALLOCATOR(exp2, type_den, env);
            when call       => return IS_VALID_CALL     (exp2, type_den, env);
            when qualified  => return IS_VALID_QUALIFIED(exp2, type_den, env);
         end case;
      end if;
   end IS_VALID_EXP2;


   procedure IS_VALID_ARRAY_COMP(exp: TREE; array_type_den: S_TYPE_DEN; env: S_ENV) return BOOLEAN is
   begin
   -- returns true, if the expression is of a type compatible with
   -- that of the components of the array type.
   end;

   procedure IS_VALID_RECORD_COMP(exp: TREE; type_den: S_TYPE_DEN; env: S_ENV) return BOOLEAN is
   begin
   -- returns true, if the expression is of a type compatible with
   -- that of the (unique) component of the record_type_den.
   end;

   procedure IS_LITERAL(exp: TREE) return BOOLEAN is
   begin
      case KIND(exp) of
         when int_number | real_number | id | string | null => return true;
         when others => return false;
      end case;
   end IS_LITERAL;

Dynamic Semantics:

   procedure EVAL_EXP(exp: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
   begin
      if IS_LITERAL(exp) and KIND(exp) /= id then
         return EVAL_LITERAL(exp, cont);
      elsif IS_NAME(exp) then
         return EVAL_NAME(exp, env, cont);
      else
         case KIND(exp) of     -- binary and unary expressions have been normalized to calls
            when membership => return EVAL_MEMBERSHIP(exp, env, cont);
            when aggregate  => return EVAL_AGGREGATE (exp, env, cont);
            when allocator  => return EVAL_ALLOCATOR (exp, env, cont);
            when call       => return EVAL_CALL      (exp, env, cont);
            when qualified  => return EVAL_QUALIFIED (exp, env, cont);
         end case;
      end if;
   end EVAL_EXP;
```

4.5 Operators and Expression Evaluation

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

Syntax:

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

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

adding_operator           ::=   +  | -  | &

unary_operator            ::=   +  | -  | not

multiplying_operator      ::=   *  | /  | mod

exponentiating_operator   ::=   **
```

Abstract Syntax:

```
binary          -> EXP   BINARY OP   EXP            -- [ EXP   BINARY OP   EXP ]
unary           -> UNARY OP   EXP                   -- [ UNARY OP   EXP ]
membership      -> EXP   MEMBERSHIP OP   TYPE RANGE -- [ EXP   MEMBERSHIP OP   TYPE RANGE ]
and             ->                                  -- [ and ]
or              ->                                  -- [ or ]
xor             ->                                  -- [ xor ]
eq              ->                                  -- [ = ]
ne              ->                                  -- [ /= ]
lt              ->                                  -- [ < ]
le              ->                                  -- [ <= ]
gt              ->                                  -- [ > ]
ge              ->                                  -- [ >= ]
plus            ->                                  -- [ + ]
minus           ->                                  -- [ - ]
cat             ->                                  -- [ & ]
mult            ->                                  -- [ * ]
div             ->                                  -- [ / ]
mod             ->                                  -- [ mod ]
exponentiation  ->                                  -- [ ** ]
not             ->                                  -- [ not ]
in              ->                                  -- [ in ]
not in          ->                                  -- [ not in ]

BINARY OP ::=   and  | or   | xor
              | eq   | ne   | lt  | le  | gt  | ge
              | plus | minus | cat
              | mult | div  | mod
              | exponentiation

UNARY OP ::= plus | minus | not

MEMBERSHIP OP ::= in | not in
```

Informal Semantics:

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.

Static Semantics:

```
procedure CHECK_BINARY(binary: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   param_assoc_s: constant TREE := PRE    (EXP1(binary), PRE(EXP2(binary), EMPTY));
   ext_name     : constant TREE := OP_NAME(BINARY_OP(binary));
begin
   return CHECK_CALL2(ext_name, param_assoc_s, type_den, env);
end CHECK_BINARY;

procedure CHECK_UNARY(unary: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   param_assoc_s: constant TREE := PRE    (EXP(unary), EMPTY);
   ext_name     : constant TREE := OP_NAME(UNARY_OP(unary));
begin
   return CHECK_CALL2(ext_name, param_assoc_s, type_den, env);
end CHECK_UNARY;

procedure CHECK_MEMBERSHIP(membership: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   tree_type_den: constant TREE_S_TYPE_DEN := CHECK_TYPE_RANGE(TYPE_RANGE(membership), env);
   exp          : constant TREE            := CHECK_EXP       (EXP(membership), TYPE(tree_type_den), env);
begin
   if IS_BOOLEAN(type_den) then
      return MAKE(membership, exp, MEMBERSHIP_OP(membership), TREE(tree_type_den));
   else
      return MEMBERSHIP_EXPRESSIONS_HAVE_TYPE_BOOLEAN;
   end if;
end CHECK_MEMBERSHIP;

procedure CHECK_TYPE_RANGE(type_range: TREE; env: S_ENV) return TREE_S_TYPE_DEN is
begin
   case KIND(type_range) of
      when constrained =>  -- [ NAME CONSTRAINT ]
         return CHECK_CONSTRAINED(type_range, env);
      when pair            -- [ EXP .. EXP ]
         | typed_pair =>   -- [ NAME range EXP .. EXP ]
         declare
            type_den   : constant S_TYPE_DEN := TYPE_OF    (type_range, env);
            typed_pair : constant TREE       := CHECK_RANGE(type_range, type_den, env);
         begin
            return TREE_TYPE(MAKE(constrained, NAME(typed_pair), PAIR(typed_pair), type_den);
            -- a range is normalized into a constrained type_den.
         end;
   end case;
end CHECK_TYPE_RANGE;
```

The definitions of the predefined operationss are given in Appendix C.

Dynamic Semantics:

Binary and unary expressions have been normalized into procedure calls.

```
procedure EVAL_MEMBERSHIP(membership: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
   -- tests constraints for type or subtype membership.
```

Dynamic Semantics:

```
procedure EVAL_CALL(call: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
   procedure CONTINUE1(param_val_s: PARAM_VAL_S) return EXEC_CONT is
      den: constant D_DEN := DEN_OF(ID(call), env);
      procedure CONTINUE2(val: VAL) return EXEC_CONT is
         type_den: constant D_TYPE_DEN := RESULT(den);
         ex_env  : constant EX_ENV     := EX_ENV(env);
      begin
         if IS_RANGE_ERROR(val, type_den) then
            return ex_env(RANGE_ERROR);
         elsif IS_OVERFLOW(val, type_den) then
            return ex_env(val, type_den);
         else return cont(val);
         end if;
      end CONTINUE2;
   begin
      return den(param_val_s, CONTINUE2);
   end CONTINUE1;
begin
   return EVAL_PARAM_ASSOC_S(PARAM_ASSOC_S(call), env, CONTINUE1);
end EVAL_CALL;
```

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.

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.

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	Operand Type 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.

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

Description:

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.

Syntax:

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

Abstract Syntax:

```
qualified -> NAME EXP  -- [ NAME (EXP) ] or [ NAME (COMP_ASSOC_S) ]
```

Static Semantics:

```
procedure CHECK_QUALIFIED(qualified: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   name_type_den: constant TREE_S_TYPE_DEN := CHECK_TYPE_MARK(NAME(qualified), env));
begin
   if not IS_COMPATIBLE(type_den, TYPE_DEN(tree_type_den)) then
      return INCOMPATIBLE_TYPES;
   elsif IS_VALID_EXP(EXP(qualified), type_den, env) then -- explicit type_den or subtype_den specification
      return MAKE(qualified, NAME(name_type_den), CHECK_EXP      (EXP(qualified), type_den, env));
   else
      return MAKE(qualified, NAME(name_type_den),CHECK_CONVERSION(EXP(qualified), type_den, env);
   end if;
end CHECK_QUALIFIED;

-- The function IS_VALID_EXP checks that the expression is of the given type.
-- It is defined in Chapter 6.
```

Dynamic Semantics:

```
procedure EVAL_QUALIFIED(qualified: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
   name: constant TREE  := NAME (qualified);
   exp : constant TREE  := EXP  (qualified);
   den : constant S_DEN :=DEN_OF(name, env);
   procedure CONTINUE(val: VAL) return EXEC_CONT is
   begin
      case KIND(exp) of
         when qualified => -- conversion from the source type "NAME(exp)" to the target type "name"
            return CONVERT(val, den, DEN_OF(NAME(exp), env), cont);
         when others => -- explicit type or subtype specification
            return TEST_CONSTRAINT(val, CONSTRAINT(den), cont);
      end case;
   end CONTINUE;
begin
   return EVAL_EXP(exp, env, CONTINUE);
end EVAL_QUALIFIED;
```

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.

Dynamic Semantics:

```
procedure TEST_CONSTRAINT(val: VAL; constraint_val: CONSTRAINT_VAL; cont: EVAL_CONT) return EXEC_CONT is
begin
-- test that the value "val" satisfies the constraint value "constraint_val" and raises
-- appropriate exception, if necessary.
end;
```

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.

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.

Static Semantics:

```
procedure CHECK_CONVERSION(exp: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
begin
   if IS_NUMERIC(type_den, env) then
      return SOLVE_CONVERSION(exp, NUMERIC_TYPE_S(env) , env);
   elsif IS_ARRAY(type_den) then
      return SOLVE_CONVERSION(exp, ARRAY_TYPE_S(type_den, env), env);
   elsif IS_ACCESS(type_den) then
      return SOLVE_CONVERSION(exp, ACCESS_TYPE_S(type_den, env), env);
   else -- conversion between objects of derivable types
      return SOLVE_CONVERSION(exp, PRE(OLD_TYPE(type_den, env), DERIVED_TYPE_S(type_den, env)), env);
   end if;
end CHECK_CONVERSION;

procedure NUMERIC_TYPE_S(env: S_ENV) return S_TYPE_DEN_S is
begin
   -- the list of all visible numeric types
end;

procedure ARRAY_TYPE_S (type_den: S_TYPE_DEN; env: S_ENV) return S_TYPE_DEN_S is
begin
   -- the list of all visible array types to which the given type_den may be converted
end;

procedure ACCESS_TYPE_S (type_den: S_TYPE_DEN; env: S_ENV) return S_TYPE_DEN_S is
begin
-- the list of all visible access types to which the given type_den may be converted
end;

procedure SOLVE_CONVERSION(exp: TREE; type_den_s: S_TYPE_DEN_S; env: S_ENV) return TREE is
   type_den_s2: constant S_TYPE_DEN_S := VALID_CONVERSION_S(exp, type_den_s, env);
```

```
      begin
         if IS_EMPTY(type_den_s2) then
            return INCOMPATIBLE_CONVERSION;
         elsif LENGTH(type_den_s2) > 1 then
            return AMBIGUOUS_CONVERSION;
         else -- the source type_den of the conversion is inserted in the result
            return MAKE(qualified, ID(HEAD(type_den_s2)), CHECK_EXP(exp, HEAD(type_den_s2), env));
         end if;
      end SOLVE_CONVERSION;

      procedure VALID_CONVERSION_S(exp: TREE; type_den_S: S_TYPE_DEN_S; env: S_ENV) return S_TYPE_DEN_S is
      begin
         if IS_EMPTY(type_den_s) then
            return EMPTY;
         elsif IS_VALID_EXP(exp, HEAD(type_den_s), env) then
            return PRE(HEAD(type_den_s), VALID_CONVERSION_S(exp, TAIL(type_den_s), env));
         else
            return VALID_CONVERSION_S(exp, TAIL(type_den_s), env);
         end if;
      end VALID_CONVERSION_S;

      -- IS_VALID_EXP is defined in Chapter 6.
```

Dynamic Semantics:

```
      procedure CONVERT(val: VAL; type_den1, type_den2: D_TYPE_DEN; cont: EVAL_CONT) return EXEC_CONT is
      begin
         -- Performs conversion from source type_den "type_den2" to target type_den "type_den1" and
         -- tests appropriate constraints
      end CONVERT;
```

4.7 Allocators

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

Syntax:

```
      allocator ::= new qualified_expression
```

Abstract SYNTAX:

```
      allocator -> QUALIFIED -- [ new QUALIFIED ]

      QUALIFIED -> qualified
```

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

Static Semantics:

```
procedure CHECK_ALLOCATOR(allocator: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   qualified    : constant TREE            := QUALIFIED      (allocator);
   name         : constant TREE            := NAME           (qualified);
   exp          : constant TREE            := EXP            (qualified);
   tree_type_den: constant TREE_S_TYPE_DEN := CHECK_TYPE_MARK(name, env);
begin
   if IS_ACCESS(type_den) and IS_COMPATIBLE(type_den, TYPE_DEN(tree_type_den)) then
      declare
         exp2: constant TREE := CHECK_EXP(exp, OBJECT_TYPE(type_den), env);
         qualified2: constant TREE := MAKE(qualified, ID(type_den), exp2);
      begin
         return MAKE(allocator, qualified2);
      end;
   else
      return INCOMPATIBLE_ACCESS_TYPE_DENS;
   end if;
end CHECK_ALLOCATOR;
```

Dynamic Semantics:

```
procedure EVAL_ALLOCATOR(allocator: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
begin
   -- allocates dynamic object, initializes it and returns location of object
   -- as access value
end EVAL_ALLOCATOR;
```

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

Static Semantics:

```
procedure CHECK_STATIC_EXP(exp: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
begin
   -- Checks whether exp is a static expression and returns a literal
   -- corresponding to its value.  If an exception (e.g., RANGE_ERROR
   -- or OVERFLOW) is raised during this evaluation, the
```

```
   -- result is an erroneous tree.
end CHECK_STATIC_EXP;
```

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.

Syntax:

```
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
```

Abstract Syntax:

```
STM_S ::= statement_s
STM   ::= assign    | call   | exit   | return | goto   | assert
        | initiate  | delay  | raise  | abort  | code   | null
        | if        | case   | loop   | accept | select | block
        | labeled

stm_s   -> STM...      -- [STM...]
labeled ->  ID   STM   -- [<<ID>> STM]
```

Description:

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.

Static Semantics:

```
procedure CHECK_STM(stm: TREE; env: S_ENV) return TREE is
begin
   case KIND(stm) of
      when assign   => return CHECK_ASSIGN  (stm, env);
      when call     => return CHECK_CALL    (stm, VOID, env);
      when exit     => return CHECK_EXIT    (stm, env);
      when return   => return CHECK_RETURN  (stm, env);
      when goto     => return CHECK_GOTO    (stm, env);
      when assert   => return CHECK_ASSERT  (stm, env);
      when initiate => return CHECK_INITIATE(stm, env);
      when delay    => return CHECK_DELAY   (stm, env);
      when raise    => return CHECK_RAISE   (stm, env);
      when abort    => return CHECK_ABORT   (stm, env);
      when code     => return CHECK_CODE    (stm, env);
      when null     => return stm;
      when if       => return CHECK_IF      (stm, env);
      when case     => return CHECK_CASE    (stm, env);
      when loop     => return CHECK_LOOP    (stm, env);
      when accept   => return CHECK_ACCEPT  (stm, env);
      when select   => return CHECK_SELECT  (stm, env);
      when block    => return CHECK_BLOCK   (stm, env, EMPTY);
      when labeled =>
         if IS_LABELED_LOOP(stm) then
            declare
               env2: constant S_ENV := DECL_MARKER(ID(stm), env);
            begin
               return MAKE(labeled, ID(stm), CHECK_STM(STM(stm), env2);
            end;
         else
            return MAKE(labeled, ID(stm), CHECK_STM(STM(stm), env));
         end if;
   end case;
end CHECK_STM;

procedure CHECK_STM_S is new CHECK_S(CHECK_STM);

procedure IS_LABELED_LOOP(stm: TREE) return BOOLEAN is
begin
   case KIND(stm) of
      when loop    => return true;
      when labeled => return IS_LABELED_LOOP(STM(stm));
      when others  => return false;
   end case;
end IS_LABELED_LOOP;
```

Dynamic Semantics:

```
procedure EXEC_STM(stm: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
begin
   case KIND(stm) of
      when assign   => return EXEC_ASSIGN(stm, env, cont);
      when call     => return EXEC_CALL  (stm, env, cont);
      when exit     => return EXEC_EXIT  (stm, env, cont);
      when return   => return EXEC_RETURN(stm, env, cont);
      when goto     => return EXEC_GOTO  (stm, env, cont);
      when assert   => return EXEC_ASSERT(stm, env, cont);
      when null     => return cont;
      when if       => return EXEC_IF    (stm, env, cont);
      when case     => return EXEC_CASE  (stm, env, cont);
      when loop     => return EXEC_LOOP  (stm, env, cont);
      when block    => return EXEC_BLOCK (stm, env, cont);
      when labelled =>
         declare
            procedure CONTINUE(env: D_ENV) return EXEC_CONT is
            begin
               return EXEC_STM(STM(stm), env, cont)
            end;
         begin
            return DECL_MARKER(ID(stm), env, CONTINUE)
         end;
   end case;
end EXEC_STM;

procedure EXEC_STM_S is new EXEC_S(EXEC_STM);
```

5.1 Assignment Statements

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

Syntax:

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

Abstract Syntax:

```
assign -> NAME   EXP    -- [ NAME := EXP; ]
```

Description:

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).

Static Semantics:

```
procedure CHECK_ASSIGN(assign: TREE; env: S_ENV) return TREE is
   den      : constant S_DEN_OF   := DEN_OF(NAME(assign), env);
   type_den: constant S_TYPE_DEN := TYPE_DEN(den);
   name     : constant TREE       := CHECK_VARIABLE(NAME(assign), type_den, env);
   exp      : constant TREE       := CHECK_EXP(EXP(assign), type_den, env);
begin
   return MAKE(assign, name, exp);
end CHECK_ASSIGN;
```

Dynamic Semantics:

```
procedure EXEC_ASSIGN(assign: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
   procedure CONTINUE1(l_val: VAL) return EXEC_CONT is
      procedure CONTINUE2(r_val: VAL) return EXEC_CONT is
         den: constant D_DEN := DEN_OF(NAME(assign), env);
         constr_den: constant D_CONSTR_DEN := CONSTR_DEN(den);
      begin
         return TEST_ASSIGN(r_val, constr_den, UPDATE(l_val, r_val, cont));
      end CONTINUE2;
   begin
      return EVAL_EXP(EXP(assign), env, CONTINUE2);
   end CONTINUE1;
begin
   return LOCATE(NAME(assign), env, CONTINUE1);
end EXEC_ASSIGN;
```

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.

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.

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.

Syntax:

```
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
```

Description:

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.

Abstract Syntax:

```
call          -> NAME  PARAM_ASSOC_S -- [NAME(PARAM_ASSOC_S)]

param_assoc_s -> PARAM_ASSOC ...     -- [PARAM_ASSOC, ...]

in_assoc      -> ID   EXP            -- [ID := EXP]
out_assoc     -> ID   EXP            -- [ID =: EXP]
in_out_assoc  -> ID   EXP            -- [ID :=: EXP]

PARAM_ASSOC_S ::= param_assoc_s
PARAM_ASSOC   ::= EXP | in_assoc | out_assoc | in_out_assoc
```

Normalization:

Each positional association is normalized into a named association.

Static Semantics:

```
procedure CHECK_CALL(call: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
begin
   return CHECK_CALL2(NAME(call), PARAM_ASSOC_S(call), type_den, env);
end CHECK_CALL;

procedure CHECK_CALL2(ext_name: TREE; param_assoc_s: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   den_s1: constant S_DEN_S := DEN_OF(ext_name, env); -- all denotations of the ext_name
   den_s2: constant S_DEN_S := VALID_OVERLOADING_S(den_s1, param_assoc_s, type_den, env);
begin
   if IS_EMPTY(den_s2) then
      return INCOMPATIBLE_CALL;
   elsif LENGTH(den_s2) > 1 then
      return AMBIGUOUS_CALL;
   else  -- solution of overloading yields a unique denotation
      declare
         den: constant S_DEN := HEAD(den_s2);
         id : constant TREE  := ID(den);  -- unique identifier associated with the subprogram declaration
                                          -- which is identified by the call
         param_s       : constant S_DEN_S := PARAM_S(den);
         param_assoc_s2: constant TREE    := NORMALIZE_PARAM_ASSOC_S(param_s, param_assoc_s, env);
         param_assoc_s3: constant TREE    := CHECK_PARAM_ASSOC_S(param_s, param_assoc_s2, env);
      begin
         return MAKE(call, id, param_assoc_s3);
      end;
   end if;
end CHECK_CALL_2;

procedure CHECK_PARAM_ASSOC_S(param_s: DEN_S; param_assoc_s: TREE; env: S_ENV) return TREE is
begin
   if IS_EMPTY(param_s) then
      return EMPTY(param_assoc_s);
   else
      PRE(CHECK_PARAM_ASSOC  (HEAD(param_s), HEAD(param_assoc_s), env),
          CHECK_PARAM_ASSOC_S(TAIL(param_s), TAIL(param_assoc_s), env));
   end if;
end CHECK_PARAM_ASSOC_S;

procedure CHECK_PARAM_ASSOC(param: DEN; param_assoc: TREE; env: S_ENV) return TREE is
begin  -- param_assoc has been normalized to [ID := EXP], [ID =: EXP], or [ID :=: EXP]
   return MAKE(KIND(param_assoc), ID(param_assoc), CHECK_EXP(EXP(param_assoc),TYPE_DEN(param), env)));
end CHECK_PARAM_ASSOC;
```

Dynamic Semantics:

```
procedure EXEC_CALL(call: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
   procedure CONTINUE(param_val_s: PARAM_VAL_S;) return EXEC_CONT is
      den: constant D_DEN := DEN_OF(ID(call), env);
   begin
      return den(param_val_s, cont)
   end CONTINUE;
begin
   return EVAL_PARAM_ASSOC_S(PARAM_ASSOC_S(call), env, CONTINUE);
end EXEC_CALL;
```

```
procedure EVAL_PARAM_ASSOC_S(param_assoc_s: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
begin
   if IS_EMPTY(param_assoc_s) then
      return cont(EMPTY)
   else
      declare
         procedure CONTINUE1(param_val: PARAM_VAL) return EXEC_CONT is
            procedure CONTINUE2(param_val_s: PARAM_VAL_S) return EXEC_CONT is
            begin
               return cont(PRE(param_val, param_val_s));
            end CONTINUE2;
         begin
            return EVAL_PARAM_ASSOC_S(TAIL(param_assoc_s), env, CONTINUE2);
         end CONTINUE1;
      begin
         return EVAL_PARAM_ASSOC(HEAD(param_assoc_s), env, CONTINUE1);
      end;
   end if;
end EVAL_PARAM_ASSOC_S;

procedure EVAL_PARAM_ASSOC(param_assoc: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
   procedure CONTINUE(val: VAL) return EXEC_CONT is
   begin
      return cont (PARAM_VAL(ID(param_assoc), NATURE(param_assoc), val));
   end CONTINUE;
begin
   case KIND(param_assoc) of
      when in_assoc =>
         return EVAL_EXP(EXP(param_assoc), env, CONTINUE);
      when out_assoc | in_out_assoc =>
         return LOCATE(EXP(param_assoc), env, CONTINUE);
   end case;
end EVAL_PARAM_ASSOC;
```

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.

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.

Syntax:

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

Abstract Syntax:

return -> EXP_VOID -- [**return** EXP_VOID;]

Description:

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.

Static Semantics:

```
procedure CHECK_RETURN(return: TREE; env: S_ENV) return TREE is
   type_den: constant S_TYPE_DEN := RESULT (CURRENT_SUBPROGRAM(env));
begin
   if not IS_IN_SUBPROGRAM(env) then
      return RETURN_ONLY_ALLOWED_IN_SUBPROGRAM_BODY;
   elsif IS_IN_ACCEPT (env) then
      return RETURN_NOT_ALLOWED_IN_ACCEPT_STATEMENT;
   elsif IS_INVALID_RETURN(EXP_VOID(return), type_den) then
      return RETURN_INCOMPATIBLE_WITH_SUBPROGRAM_RESULT;
   else
      case KIND(EXP_VOID(return)) of
         when void   => return return;
         when others => return MAKE(return, CHECK_EXP(EXP_VOID(return), type_den, env));
      end case;
   end if;
end CHECK_RETURN;


procedure IS_INVALID_RETURN(exp_void: TREE; type_den: S_TYPE_DEN;) return BOOLEAN is
begin
   case KIND(exp_void) of
      when void   => return type_den /= VOID;
      when others => return type_den = VOID;
   end case;
end IS_VALID_RETURN;
```

Dynamic Semantics:

```
procedure EXEC_RETURN(return: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
begin
   case KIND(EXP_VOID(stm)) of
      when void   => RETURN_EXEC_CONT(env));
         -- RETURN_EXEC_CONT(env) is the continuation of the current procedure call
      when others => EVAL_EXP(EXP_VOID(stm), env, RETURN_EVAL_CONT(env));
         -- RETURN_EVAL_CONT(env) is the continuation of the current
         -- function or value returning procedure call
   end case;
end EXEC_RETURN_STM;
```

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.

Syntax:

```
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}
```

Abstract Syntax:

```
if            -> CONDITIONAL_S   STM_LIS     -- [if CONDITIONAL_S else STM_S end if;]
conditional_s -> CONDITIONAL...              -- [CONDITIONAL elsif ... ]
conditional   -> COND   STM_S                -- [COND then STM_S]
condition     -> EXP    CONDITION_OP   COND  -- [EXP CONDITION_OP COND]
and_then      -> EXP    CONDITION_OP   COND  -- [and then]
or_else       -> EXP CONDITION_OP COND       -- [or else]

CONDITIONAL_S ::= conditional_s
CONDITIONAL   ::= conditional
COND          ::= EXP | condition
CONDITION_OP  ::= and_then | or_else
```

Description:

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.

Normalization:

```
[if CONDITIONAL_S else STM_S end if;] ->
[if CONDITIONAL_S elsif true then STM_S end if;]
```

Static Semantics:

```
procedure CHECK_IF(if: TREE; env: S_ENV) return TREE is
   env2: S_ENV := LABELS_IN(CONDITIONAL_S(if), env);
begin
   return MAKE(if, CHECK_CONDITIONAL_S(CONDITIONAL_S(if), env2), EMPTY);
end CHECK_IF;

procedure CHECK_CONDITIONAL_S is new CHECK_S(CHECK_CONDITIONAL);

procedure CHECK_CONDITIONAL(conditional: TREE; env: S_ENV) return TREE is
begin
```

```
    return MAKE(conditional, CHECK_COND(COND(conditional), env), CHECK_STM_S(STM_S(conditional), env));
end CHECK_CONDITIONAL;
```

Dynamic Semantics:

```
procedure EXEC_CONDITIONAL(conditional: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
   procedure CONTINUE(val: VAL) return EXEC_CONT is
   begin
      if BOOLEAN(val) then
         return EXEC_STM_S(STM_S(conditional), env, cont);
      else
         return cont;
      end if;
   end CONTINUE;
begin
   return EVAL_COND(COND(conditional), env, CONTINUE);
end EXEC_CONDITIONAL;
```

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.

Static Semantics:

```
procedure CHECK_COND(cond: TREE; env: S_ENV) return TREE is
begin
   if IS_EXP(cond) then  -- [ EXP ]
      return CHECK_EXP(cond, BOOLEAN_TYPE, env);
   else  -- [EXP CONDITION_OP COND ]
      declare
         exp  : TREE := CHECK_EXP(EXP(cond), BOOLEAN_TYPE, env);
         cond2: TREE := CHECK_COND(COND(cond), env);
      begin
         return MAKE(condition, exp, CONDITION_OP(cond), cond2);
      end;
   end if;
end CHECK_COND;
```

Dynamic Semantics

```
procedure EVAL_COND(cond: TREE; env: D_ENV: cont: EVAL_CONT) return EXEC_CONT is
begin
   if IS_EXP(cond) then  -- [EXP]
      return EVAL_EXP(cond, env, cont);
   else                  -- [EXP CONDITION_OP COND ]
      return EVAL_CONDITION(cond, env, cont);
   end if;
end EVAL_COND;



procedure EVAL_CONDITION(condition: TREE; env: D_ENV; cont: EVAL_CONT) return EXEC_CONT is
   procedure CONTINUE1(val: VAL) return EXEC_CONT is
   begin
      if BOOLEAN(val) then
         return EVAL_COND(COND(condition), env, cont);
      else
         return cont(val);
      end if;
   end CONTINUE1;
   procedure CONTINUE2(val: VAL) return EXEC_CONT is
   begin
      if BOOLEAN(val) then
         return cont(val);
      else
         return EVAL_COND(COND(condition), env, cont);
      end if;
   end CONTINUE2;
begin
   case CONDITION_OP(condition) of
      when and then => return EVAL_EXP(EXP(exp), env, CONTINUE1);
      when or else  => return EVAL_EXP(EXP(exp), env, CONTINUE2);
   end case;
end EVAL_CONDITION;
```

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.

Syntax:

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

Abstract Syntax:

```
case          -> EXP   ALTERNATIVE_S  -- [ case EXP of ALTERNATIVE_S end case; ]
alternative_s -> ALTERNATIVE ...      -- [ ALTERNATIVE ... ]
alternative   -> CHOICE_S STM_S       -- [ when CHOICE_S => STM_S ]

ALTERNATIVE_S ::= alternative_s
ALTERNATIVE   ::= alternative
```

Description:

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.

Static Semantics:

```
procedure CHECK_CASE(case: TREE; env: S_ENV) return TREE is
   env2         : constant S_ENV        := LABELS_IN(ALTERNATIVE_S(case), env);
   type_den     : constant S_TYPE_DEN   := TYPE_OF(EXP(case), env2);
   exp          : constant TREE         := CHECK_EXP(EXP(case), type_den, env2);
   alternative_s: constant ALTERNATIVE_S := CHECK_ALTERNATIVE_S(ALTERNATIVE_S(case), type_den, env2);
begin
   if not IS_DISCRETE(type_den) then
      return TYPE_MUST_BE_DISCRETE;
   elsif not IS_EXHAUSTIVE(alternative_s, type_den) then
      return CHOICE_S_MUST_BE_EXHAUSTIVE;
   else
      return MAKE(case, exp, alternative_s);
   endif;
end CHECK_CASE;


procedure CHECK_ALTERNATIVE_S is new CHECK_S2(S_TYPE_DEN, CHECK_ALTERNATIVE);

procedure CHECK_ALTERNATIVE(alternative: TREE; type_den: S_TYPE_DEN; env: S_ENV) return TREE is
   choice_s1: constant TREE := CHECK_CHOICE_S(CHOICE_S(alternative), type_den, env);
   choice_s2: constant TREE := NORMALIZE_CHOICE_S(choice_s1, type_den);
                           -- normalizes each choice such that a range is
                           -- replaced by a list of choice values
   stm_s: constant TREE     := CHECK_STM_S(STM_S(alternative), env);
begin
   return MAKE(alternative, choice_s2, stm_s);
end CHECK_ALTERNATIVE;
```

Dynamic Semantics:

```
procedure EXEC_CASE(case: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
   env2: constant D_ENV := LABELS_IN(ALTERNATIVE_S(case), env);
   procedure CONTINUE(val: VAL) return EXEC_CONT is
   begin
      return EXEC_ALTERNATIVE_S(ALTERNATIVE_S(case), val, env2, cont);
   end CONTINUE;
begin
   return EVAL_EXP(EXP(case), env2, CONTINUE);
end EXEC_CASE;

procedure EXEC_ALTERNATIVE_S is new EXEC_S2(VAL, EXEC_ALTERNATIVE_S);


procedure EXEC_ALTERNATIVE(alternative: TREE; val: VAL; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
   choice_s: constant TREE := CHOICE_S(alternative);
   stm_s   : constant TREE := STM_S(alternative);
begin
   if IS_EMPTY(choice_s) then
      return cont;
   else
      declare
         alternative2: constant TREE := MAKE(alternative, TAIL(choice_s), stm_s);
         procedure CONTINUE (val2: VAL) return EXEC_CONT is
         begin
            if val = val2 or val2 = others_val then
               return EXEC_STM_S(STM_S(alternative), env, cont);
            else
               return EXEC_ALTERNATIVE(alternative2, env, cont);
            end if;
         end;
      begin
         return EVAL_EXP(HEAD(choice_s), env, CONTINUE);
      end;
   end if;
end EXEC_ALTERNATIVE;
```

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.

Syntax:

```
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
```

Abstract Syntax:

```
loop      -> ITERATION   STM_S   -- [ ITERATION loop STM_S end loop; ]
for       -> ID   RANGE          -- [ for ID in RANGE ]
reverse   -> ID   RANGE          -- [ for ID in reverse RANGE]
while     -> EXP                 -- [ while EXP ]

ITERATION ::= for | reverse | while | void
```

Description:

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**.

Static Semantics:

```
procedure CHECK_LOOP(loop: TREE; env: S_ENV) return TREE is
   env2: constant S_ENV := IN_LOOP(env);
   env3: constant S_ENV := LABELS_IN_STM_S(STM_S(loop), env2);
   iteration_env     : constant TREE_ENV := CHECK_ITERATION(ITERATION(loop), env);
   stm_s             : constant TREE     := CHECK_STM_S(STM_S(loop), ENV(iteration_env));
begin
   return MAKE(loop, TREE(iteration_env), stm_s);
end CHECK_LOOP;

procedure CHECK_ITERATION(iteration: TREE; env: S_ENV) return TREE_ENV is

begin
   case KIND(iteration) of
      when for            -- [ for ID in RANGE ]
```

```
            | reverse =>    -- [ for ID in reverse RANGE ]
            declare
               type_den  : constant S_TYPE_DEN := TYPE_OF(RANGE(iteration), env);
               range     : constant TREE       := CHECK_RANGE(RANGE(iteration), type_den, env);
               desden    : constant S_DESDEN   := DESCRIPTION(type_den, EMPTY);
               id_env    : constant TREE_ENV   := DECL(ID(iteration), constant, desden, EMPTY);
               env2      : constant S_ENV      := NESTED_ENV(env, ENV(id_env));
               iteration2: constant TREE       := MAKE(KIND(iteration), ID(id_env), range);
            begin
               return TREE_ENV(iteration2, env2);
            end;

            when while =>  -- [ while EXP ]
               declare
                  exp       : constant TREE := CHECK_EXP(EXP(iteration), BOOLEAN_TYPE, env);
                  iteration2: constant TREE := MAKE(while, exp);
               begin
                  return TREE_ENV(iteration2, env);
               end;
            when void =>
               return TREE_ENV(void, env);
         end case;
      end CHECK_ITERATION;


Dynatic Semantics:

      procedure EXEC_LOOP(loop: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
         env2: constant D_ENV := SET_LOOP(env, cont);
         env3: constant D_ENV := LABELS_IN_STM_S(STM_S(loop), env2, cont);
      begin
         case KIND(ITERATION(loop)) of
            when for     =>  -- [ for ID in RANGE loop STM_S end loop; ]
               return EXEC_FOR_LOOP(loop, env3, cont);
            when reverse =>  -- [ for ID in reverse RANGE loop STM_S end loop; ]
               return EXEC_REVERSE_LOOP(loop, env3, cont);
            when while   =>  -- [ while EXP loop STM_S end loop; ]
               return EXEC_WHILE_LOOP(loop, env3, cont);
            when void    =>  -- [ loop STM_S end loop; ]
               return EXEC_STM_S(STM_S(loop), env, EXEC_LOOP(loop, env, cont));
         end case;
      end EXEC_LOOP;


      procedure EXEC_FOR_LOOP(loop: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
         iteration: constant TREE := ITERATION(loop);
         type_den : constant D_TYPE_DEN := TYPE_OF(NAME(RANGE(iteration)), env);
         procedure CONTINUE1(val1, val2: VAL) return EXEC_CONT is
            procedure CONTINUE2(val: VAL) return EXEC_CONT is
               procedure CONTINUE3(env2: D_ENV) return EXEC_CONT is
               begin
                  return EXEC_STM_S(STM_S(loop), env2, CONTINUE2(SUCC(val, type_den)));
               end CONTINUE3;
```

```
            begin
               if LE(val, val2, type_den) then
                  return DECL(ID(iteration), constant, val, env, CONTINUE3);
               else
                  return cont;
               end if;
            end CONTINUE2;

         begin
            return CONTINUE2(val1);
         end CONTINUE1;
      begin
         return EVAL_RANGE(RANGE(iteration), env, CONTINUE1);
      end EXEC_FOR_LOOP;


      procedure EXEC_REVERSE_LOOP(loop: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
         iteration: constant TREE       := ITERATION(loop);
         type_den : constant D_TYPE_DEN := TYPE_OF(NAME(RANGE(iteration)), env);
         procedure CONTINUE1(val1, val2: VAL) return EXEC_CONT is
            procedure CONTINUE2(val: VAL) return EXEC_CONT is
               procedure CONTINUE3(env2: D_ENV) return EXEC_CONT is
               begin
                  return EXEC_STM_S(STM_S(loop), env2, CONTINUE2(PRED(val, type_den)));
               end CONTINUE3;
            begin
               if LE(val2, val, type_den) then
                  return DECL(ID(iteration), constant, val, env, CONTINUE3);
               else
                  return cont;
               end if;
            end CONTINUE2;
         begin
            return CONTINUE2(val2);
         end CONTINUE1;
      begin
         return EVAL_RANGE(RANGE(iteration), env, CONTINUE1);
      end EXEC_REVERSE_LOOP;


      procedure EXEC_WHILE_LOOP(loop: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is

         procedure CONTINUE(val: VAL) return EXEC_CONT is
         begin
            if BOOLEAN(val) then
               return EXEC_STM_S(STM_S(loop), env, EXEC_WHILE_LOOP(loop, env, cont);
            else
               return cont;
            end if;
         end CONTINUE;

      begin
         return EVAL_EXP(EXP(loop), env, CONTINUE);
      end EXEC_WHILE_LOOP;
```

5.7 Exit Statements

An exit statement causes explicit termination of an enclosing loop.

Syntax:

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

Abstract Syntax:

```
exit      -> ID_VOID  COND_VOID   -- [ exit ID_VOID when COND; ]
                                  -- [ exit ID_VOID; ]
COND_VOID ::= COND | void
```

Description:

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.

Normalization:

```
[exit ID_VOID when COND;] ->
[ if COND then exit ID_VOID end if; ]
```

Static Semantics:

```
procedure CHECK_EXIT(exit: TREE; env: S_ENV) return TREE is
begin
   if IS_IN_LOOP (env) then
      case KIND(ID_VOID(exit)) of
         when void =>  -- [ exit; ]
            return exit;
         when id   =>  -- [ exit ID; ]
            if IS_MARKER(ID_VOID(exit), env) then
               return exit;
            else
               return EXIT_IDENTIFIER_MUST_BE_VISIBLE_LOOP_LABEL;
            end if;
      end case;
   else
      return EXIT_ONLY_FROM_WITHIN_LOOPS;
   end if;
end CHECK_EXIT;
```

Dynamic Semantics:

```
procedure EXEC_EXIT(exit: TREE; env: D_ENV; cont: EXEC_CONT) return EXEC_CONT is
begin
   case KIND(ID_VOID(exit)) of
      when void =>  -- [ exit; ]
         return LOOP_CONT(env);  -- the continuation of the current loop
```

```
        when id   =>  -- [ exit ID; ]
            return MARKER_VAL(ID_VOID(exit), env);
    end case;
end EXEC_EXIT;
```

5.8 Goto Statements

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

Syntax:

```
goto_statement ::= goto identifier;
```

Abstract Syntax:

```
goto -> ID  -- [ goto ID; ]
```

Description:

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.

Static Semantics:

```
procedure CHECK_GOTO(goto: TREE; env: D_ENV) return TREE is
begin
   if IS_LABEL(ID(goto), env) then
      return goto;
   else
      return IDENTIFIER_IN_GOTO_MUST_BE_VISIBLE_LABEL;
   end if;
end CHECK_GOTO;
```

Dynamic Semantics:

```
procedure EXEC_GOTO(goto: TREE; env: D_ENV, cont: EXEC_CONT) return EXEC_CONT is
begin
   return LABEL_CONT(ID(goto), env);
end EXEC_GOTO;
```

5.9 Assert Statement

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

Syntax:

```
assert_statement ::= assert condition;
```

Abstract Syntax:

```
assert -> COND  -- [ assert COND; ]
```

Description:

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).

Static Semantics:

```
procedure CHECK_ASSERT(assert: TREE; env: S_ENV) return TREE is
begin
   return MAKE(assert, CHECK_COND(COND(assert), BOOLEAN_TYPE, env);
end CHECK_ASSERT;
```

Dynamic Semantics:

```
procedure EXEC_ASSERT(assert: TREE; env:D_ENV; cont: EXEC_CONT) return EXEC_CONT is
   procedure CONTINUE(val: VAL) return EXEC_CONT is
   begin
      if BOOLEAN(val) then
         return cont;
      else
         return EXEC_RAISE(assert_error, env)
      end if;
   end CONTINUE;
begin
   return EVAL_COND(COND(assert), env, CONTINUE);
end EXEC_ASSERT;
```

5.10 Auxiliary Definitions for Labels

Normalization:

The following functions are used to ensure that labels are not multiply defined within subprograms or modules.

```
procedure LABELS_IN_STM(stm: TREE; label_s: TREE) return TREE
begin
   case KIND(stm) of
      when labeled => -- [ << ID >> STM]
         if IS_IN(ID(stm), label_s) then
```

```
                return ERROR_LABELS_MUST_BE_UNIQUE;
             else
                return LABELS_IN_STM(STM(stm), PRE(ID(stm), label_s));
             end if;
          when if       =>  -- [ if CONDITONAL_S end if; ]
             return LABELS_IN(CONDITIONAL_S(stm), label_s);
          when case     =>  -- [case EXP of ALTERNATIVE_S end case; ]
             return LABELS_IN(ALTERNATIVE_S(stm), label_s);
          when loop     -- [ ITERATION loop STM_S end loop; ]
             accept =>  -- [ accept NAME (DECL_S) do STM_S end; ]
             return LABELS_IN_STM_S(STM_S(stm), label_s);
          when select   =>  -- [ select ALTERNATIVE_S else STM_S end if; ]
             return LABELS_IN_STM_S(STM_S(stm), LABELS_IN(ALTERNATIVE_S(stm), label_s));
          when block    =>          -- [declare DECL_PART begin STM_S exception ALTERNATIVE_S end; ]
             return LABELS_IN(ALTERNATIVE_S(stm), LABELS_IN_STM_S(STM_S(stm), label_s));
          when others   =>
             return label_s;
       end case;
    end LABELS_IN_STM;


    procedure LABELS_IN_STM_S(stm_s: TREE; label_s: TREE) return TREE is
    begin
       if IS_EMPTY(stm_s) then
          return label_s;
       else
          return LABELS_IN_STM_S(TAIL(stm_s), LABELS_IN_STM(HEAD(stm_s), label_s));
       end if;
    end LABELS_IN_STM_S;

    procedure LABELS_IN(list: TREE; label_s: TREE) return TREE is
    begin  -- list is conditional_s or alternative_s
       if IS_EMPTY(list) then
          return label_s;
       else
          return LABELS_IN(TAIL(list), LABELS_IN_STM_S(STM_S(HEAD(list)), label_s));
       end if;
    end LABELS_IN;
```

Static Semantics:

The following functions are used to declare labels local to a compound statement, block, subprogram, module, accept statement or exception handler to ensure the proper visibility of labels (see 5.6, 5.7).

```
    procedure LABELS_IN_STM(stm_s: TREE; env: S_ENV) return S_ENV is
    begin
       case KIND(stm) of
          when labeled =>  -- [<<ID>> STM ]
             return LABELS_IN_STM(STM(stm), DECL_LABEL(ID(stm), env));
          when others =>
             return env;
       end case
    end LABELS_IN_STM;
```

```
procedure LABELS_IN_STM_S(stm: TREE; env: S_ENV) return S_ENV is
begin
   if IS_EMPTY(stm_s) then
      return env;
   else
      return LABELS_IN_STM_S(TAIL(stm_s), LABELS_IN_STM(HEAD(stm_s), env));
   end if;
end LABELS_IN_STM_S;

procedure LABELS_IN(list: TREE; env: S_ENV) return S_ENV is
begin  -- list is conditional_s or alternative_s
   if IS_EMPTY(list) then
      return env;
   else
      return LABELS_IN(TAIL(list), LABELS_IN_STM_S(STM_S(HEAD(list)), label_s));
   end if;
end LABELS_IN;
```

Dynamic Semantics:

The following functions are used to associate a continuation with each label in the dynamic environment.

<to be inserted>

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.

Syntax:

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

body ::= [visibility_restriction] unit_body | body_stub

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

Abstract Syntax:

```
use         -> NAME S    ITEM S     -- [use NAME S; ITEM S]
item s      -> ITEM...              -- [ITEM; ...]
unit        -> SPECIFIER BODY       -- [SPECIFIER is BODY]

DECL PART ::= use | item s
ITEM S    ::= item s
ITEM      ::= address | decl | component rep | packing | record rep | restricted |
              subunit | EXP
BODY      ::= block   | stub | void | EXT NAME
SPECIFIER ::= subprogram | module | void | generic | family
```

Description:

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).

Static Semantics:

```
procedure CHECK_DECL_PART(decl_part: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
begin
   case KIND(decl_part) of
      when use    => return CHECK_USE(decl_part, env);
      when item_s => return CHECK_ITEM_S(decl_part, env, local);
   end case;
end CHECK_DECL_PART;


procedure CHECK_ITEM_S(item_s: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
begin
   if IS_EMPTY(item_s) then
      return TREE_ENV(item_s, local);
   else
      declare
         tree_env : constant TREE_ENV := CHECK_ITEM  (HEAD(item_s), env, local);
         local2   : constant S_ENV    := ENV         (tree_env);
         env2     : constant S_ENV    := NESTED_ENV  (env, local2);
         tree_env2: constant TREE_ENV := CHECK_ITEM_S(TAIL(item_s), env2, local2);
      begin
         return TREE_ENV(PRE(TREE(tree_env), TREE(trec_env2));, ENV(tree_env2));
      end;
   end if;
end CHECK_ITEM_S;


procedure CHECK_ITEM(item: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
begin
   case KIND(item) of
      when decl       =>    -- [NATURE  DESIGNATOR_S  DESCRIPTION; ]
                            return CHECK_DECL(item, env, local);
      when restricted =>    -- [restricted NAME_S  DECL_S]
                            return CHECK_RESTRICTED(item, env, local);
      when subunit    =>    -- [separate DECL;]
                            return CHECK_SUBUNIT(item, env, local);
      when others     =>    -- ignored in this version of static semantics.
                            return TREE_ENV(item, local);
   end case;
end CHECK_ITEM;


procedure DECL_UNIT(decl: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
   designator_s: constant TREE := DESIGNATOR_S(decl);
```

```
      nature      : constant TREE := NATURE      (decl)
      unit        : constant TREE := DESCRIPTION (decl)
   begin
      case KIND(nature) of
         function  |  -- [procedure DESIGNATOR_S DESCRIPTION]
         procedure |  -- [procedure DESIGNATOR_S DESCRIPTION]
         entry     |  -- [entry     DESIGNATOR_S DESCRIPTION]
         package      -- [package   DESIGNATOR_S DESCRIPTION]
         =>
         declare
            tree_des_den : constant TREE_DES_DEN := CHECK_UNIT(nature, unit, env);
            unit2        : constant TREE         := TREE(tree_des_den);
            den          : constant S_DEN        := DEN(nature, DES_DEN(tree_des_den));
            tree_env     : constant TREE_ENV     := CHECK_DESIGNATOR_S(designator_s, local, den);
            designator_s2: constant TREE         := TREE(tree_env);
         begin
            return TREE_ENV(MAKE(decl, nature, designator_s2), ENV(tree_env));
         end;
      end case;
   end DECL_UNIT;

   procedure CHECK_UNIT(nature: TREE; unit: TREE; env: S_ENV) return TREE_DES_DEN is
      tree_specif_den    : constant TREE_SPECIF_DEN := CHECK_SPECIFIER(SPECIFIER(unit), env, EMPTY);
      specifier          : constant TREE            := TREE(tree_specif_den);
      local              : constant S_ENV           := LOCAL_ENV(SPECIF_DEN(tree_specif_den));
   begin
      if IS_FUNCTION(nature) and SIDE_EFFECTS(BODY(unit), env, local) then
         return NO_SIDE_EFFECTS_IN_FUNCTION_BODY;
      else
         declare
            env2          : constant S_ENV           := NESTED_ENV(env, local);
            tree_body_den: constant TREE_BODY_DEN    := CHECK_BODY(BODY(unit), env2, local);
            body          : constant TREE            := TREE(tree_body_den);
            des_den       : constant DES_DEN         := DES_DEN(SPECIF_DEN(tree_specif_den),
                                                                BODY_DEN(tree_body_den));
         begin
            return TREE_DES_DEN(MAKE(unit, specifier, body), des_den);
         end;
      end if;
   end CHECK_UNIT;

   procedure CHECK_SPECIFIER(specifier: TREE; env: S_ENV; local: S_ENV) return TREE_SPECIF_DEN is
   begin
      case KIND(specifier) of
         when subprogram =>     -- [(DECL_S) RESULT]
            CHECK_SUBPROGRAM(specifier, env, local);
         when module     =>     -- [DECL_PART private DECL_PART]
            CHECK_MODULE(specifier, env, local);
         when void       =>
            return TREE_SPECIF_DEN(void, EMPTY)
         when generic    =>     -- [(DECL_S) SPECIFIER]
            CHECK_GENERIC(specifier, env, local);
         when family     =>     -- [(RANGE) SPECIFIER]
            CHECK_FAMILY(specifier, env, local);
      end case;
```

```
end CHECK_SPECIFIER;
```

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.

Syntax:

```
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 ::= procedure | 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
```

Abstract Syntax:

```
decl -> NATURE   DESIGNATOR S   DESCRIPTION
unit -> SPECIFIER   BODY
subprogram -> DECL S   RESULT  -- [(DECL S)   RESULT]

RESULT      ::= constrainted  | void
DESCRIPTION ::= instantiation | object  | renaming  | type description | unit | void
SPECIFIER   ::= subprogram    | module  | void      | generic           | family
BODY        ::= block         | stub    | void      | EXT_NAME
NATURE      ::= constant      | entry   | exception | function          | in   | in out |
                out           | package | procedure | subtype           | task | type |
                variable.
```

Description:

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.

Normalization:

```
procedure NORMALIZE_SPECIFIER(specifier: TREE) return TREE is
begin
-- this function checks that "a parameter declaration or constraint on the result
-- cannot contain an identifier declared in another parameter declaration of
-- the same formal-part".
end NORMALIZE_SPECIFIER;
```

Static Semantics:

```
procedure CHECK_SUBPROGRAM(subprogram: TREE; env: S_ENV; local: S_ENV) return TREE_SPECIF_DEN is
   env2          : constant S_ENV          := IN_SUBPROGRAM(env);
   tree_env      : constant TREE_ENV       := CHECK_DECL_S (DECL_S(subprogram), env2, local);
   decl_s        : constant TREE           := TREE         (tree_env);
   tree_type_den: constant TREE_TYPE_DEN := CHECK_RESULT (RESULT(subprogram), env2);
   result        : constant TREE           := TREE         (tree_env);
   specif_den    : constant S_SPECIF_DEN   := SPECIF_DEN   (decl_s, ENV(tree_env), TYPE_DEN(tree_type_den));
begin
   return TREE_SPECIF_DEN(MAKE(subprogram, decl_s, result), specif_den);
end CHECK_SUBPROGRAM;

procedure CHECK_RESULT(result: TREE; env: S_ENV) return TREE_TYPE_DEN is
begin
   case KIND(result) of
      when void        => return TREE_TYPE_DEN(result, TYPE_DEN(EMPTY));
      when constrained => return CHECK_TYPE   (result, TYPE_DEN(env));
   end case;
end CHECK_RESULT;
```

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.

6.4 Subprogram Bodies

Syntax:

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];
```

Abstract Syntax:

```
decl -> NATUPE      DESIGNATOR S   DESCRIPTION
init -> SPECIFIER   BODY               -- [SPECIFIER is BODY]

DESCRIPTION ::= instantiation | object | renaming | type description | unit | void
BODY        ::= block         | stub   | void     | EXT NAME
```

Description:

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.

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.

Normalization:

```
procedure NORMALIZE_FUNCTION_SPECIFIER(specifier: TREE) return TREE is
begin
-- this function checks "A function can only have in parameters
-- and must contain a return clause."
-- "A value return_ing procedure can only have in
-- parameters, its declaration must contain a return clause".
end NORMALIZE_FUNCTION_SPECIFIER;


procedure SIDE_EFFECTS(body: TREE; env: S_ENV; local: S_ENV) return BOOLEAN is
begin
-- <to be defined>
end SIDE_EFFECTS;
```

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.

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 /=.

6.7 Blocks

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

Syntax:

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

Abstract Syntax:

```
block -> DECL_PART   STM_S   ALTERNATIVE_S   -- [declare DECL_PART begin STM_S exception ALTERNATIVE_S end]
```

Description:

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.

Static Semantics:

```
procedure CHECK_BLOCK(block: TREE; env: S_ENV; local: S_ENV) return TREE is
   tree_env: constant TREE_ENV := CHECK_DECL_PART(DECL_PART(block), env, local);
begin
   if BODIES_PROVIDED(ENV(tree_env)) then
      declare
```

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.

Syntax:

```
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];
```

```
            env2          : constant S_ENV := LABELS_IN_STM_S(STM_S(block), ENV(tree_env));
            env3          : constant S_ENV := NESTED_ENV     (env, env2);
            stm_s         : constant TREE  := CHECK_STM_S    (STM_S(block), env3);
            alternative_s: constant TREE  := CHECK_HANDLER_S(HANDLER_S(block), env3);
         begin
            return MAKE(block, TREE(tree_env), stm_s, alternative_s);
         end;
      else MISSING_BODIES;
      end if;
   end CHECK_BLOCK;

   procedure BODIES_PROVIDED(env: S_ENV) return BOOLEAN is
   begin
   -- this function verifies that the body of a subprogram or module
   -- declared in the declarative part of a block or subprogram must
   -- is provided in the same declarative_part.
   end BODIES_PROVIDED;
```

Abstract Syntax:

```
decl   -> NATURE      DESIGNATOR_S   DESCRIPTION
unit   -> SPECIFIER   BODY
module -> DECL_PART   DECL_PART        -- [is DECL_PART private DECL_PART]
family -> RANGE       SPECIFIER        -- [(RANGE) SPECIFIER]

SPECIFIER   ::= subprogram   | module | void   | generic          | family
BODY        ::= block        | stub   | void   | EXT_NAME
DESCRIPTION ::= instantiation | object | renaming | type_description | unit | void
```

Description:

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.

Static Semantics:

```
procedure CHECK_MODULE(module: TREE; env: S_ENV; local: S_ENV) return TREE_SPECIF_DEN is
   tree_env  : constant TREE_ENV      := CHECK_DECL_PART(DECL_PART1(module), env, local);
   env2      : constant S_ENV         := NESTED_ENV      (env, ENV(tree_env));
   tree_env2 : constant TREE_ENV      := CHECK_DECL_PART(DECL_PART2(module), env2, ENV(tree_env));
   local2    : constant S_ENV         := REMOVE          (ENV(tree_env), local);
   specif_den: constant S_SPECIF_DEN := SPECIF_DEN      (VISIBLE(local2), ENV(tree_env2));
begin
   return TREE_SPECIF_DEN(MAKE(module, TREE(tree_env1), TREE(tree_env2)), specif_den);
end CHECK_MODULE;

-- the constraint "the batches of the specified units must appear within
-- the declarative part of the module body" is checked by CHECK_BLOCK (see Section 6.7).

procedure REMOVE(env1, env2: S_ENV) return S_ENV is
begin
-- This procedure builds a new environment of type S_ENV that
-- includes designators declared in env1 and not in env2.
-- It is used to build the environment defined by the visible part
-- of the module.
end REMOVE;
```

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.

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.

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.

Syntax:

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

Abstract Syntax:

```
restricted private ->
```

Description:

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.

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).

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.

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.

8.3 Restricted Program Units

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

Syntax

```
visibility_restriction ::= restricted [visibility_list]

visibility_list ::= (unit_name {, unit_name})
```

Abstract Syntax:

```
restricted -> NAME_S DECL_S      -- [ restricted NAME_S DECL_S ]
name_s     -> NAME ...           -- [ NAME , ... ]

NAME_S     ::= name_s
```

Description:

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.

Static Semantics:

```
function CHECK_RESTRICTED(restricted: TREE; env: S_ENV; local: S_ENV) return TREE_ENV is
   tree_env   : constant TREE_ENV := CHECK_RESTRICT_NAME(NAME_S(restricted), env);
   name_s     : constant TREE     := TREE(tree_env);
   env2       : constant S_ENV    := ENV (tree_env);
```

```
      tree_env2  : constant TREE_ENV := CHECK_DECL(DECL_S(restricted), env2, local);
      decl_s     : constant TREE     := TREE(tree_env2);
      local2     : constant S_ENV    := ENV (tree_env2);
   begin
      return TREE_ENV(MAKE(restricted, name_s, decl_s), local2);
   end;

   procedure CHECK_RESTRICT_NAME(name_s: TREE; env: S_ENV) return TREE_ENV is
   begin
      if IS_EMPTY(name_s) then
         return TREE_ENV(name_s, EMPTY);
      elsif IS_ENCLOSING(HEAD(name_s), env) then
         declare
         env2: constant S_ENV := RESTRICT_ENV(HEAD(name_s), env);
         begin
         return CHECK_RESTRICT_NAME2(TAIL(name_s), env2);
         end;
      else return CHECK_RESTRICT_NAME2(name_s, env);
      end if;
   end CHECK_RESTRICT_NAME;

   procedure RESTRICT_ENV(name: TREE; env: S_ENV) return S_ENV is
   begin
   -- This function hides everything but the entities declared
   -- within the enclosing unit specified by "name".
   end RESTRICT_ENV;

   procedure CHECK_RESTRICT_NAME2(name_s: TREE; env: S_ENV) return S_ENV is
   begin
      if IS_EMPTY(name_s) then
         return TREE_ENV(name_s, env);
      elsif IS_MODULE(DEN_OF(HEAD(name_s), env)) then
         declare
         env2     : constant S_ENV    := RESTRICTED_OP(HEAD(name_s), env);
         tree_env: constant TREE_ENV := CHECK_RESTRICT_NAME2(TAIL(name_s), env2);
         begin
         return TREE_ENV(PRE(HEAD(name_s), TREE(tree_env)),
                         ENV(tree_env));
         end;
      else return TREE_ENV(MODULE_NAME_EXPECTED, env);
   end CHECK_RESTRICT_NAME2;

   function RESTRICTED_OP(name: TREE; env: S_ENV) return S_ENV is
   begin
   -- This function makes the name "name" visible in the environment "env".
   end RESTRICTED_OP;
```

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.

Syntax:

```
use_clause ::= use module_name {, module_name};
```

Abstract Syntax:

```
use    -> NAME_S  ITEM_S     -- [ use NAME_S; ITEM_S ]
name_s -> NAME ...           -- [ NAME , ... ]
```

Description:

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.

Static Semantics:

```
procedure CHECK_USE(use: TREE, env: S_ENV) return TREE_ENV is:
   tree_env1: constant TREE_ENV := USE_ENV(NAME_S(use), env);
   name_s   : constant TREE     := NAME_S       (tree_env1);
   env1     : constant S_ENV    := ENV          (tree_env1);
   tree_env2: constant TREE_ENV := CHECK_ITEM_S(ITEM_S(use), env2, EMPTY);
   item_s   : constant TREE     := TREE         (tree_env2);
   env2     : constant S_ENV    := ENV          (tree_env2);
begin
   return TREE_ENV(MAKE(use, name_s, item_s), env2);
end CHECK_USE;

procedure USE_ENV(name_s: TREE; env: S_ENV) return TREE_ENV is
begin
   if IS_EMPTY(name_s) then
      return TREE_ENV(name_s, env);
   elsif
      IS_MODULE(DEN_OF(HEAD(name_s))) then
      declare
         specif_den: constant S_SPECIF_DEN := SPECIF_DEN(DEN_OF(HEAD(name_s), env));
         env2      : constant S_ENV        := USE_OP(env, VISIBLE(specif_den));
         tree_env  : constant TREE_ENV     := USE_ENV(TAIL(name_s), env2);
      begin
         return TREE_ENV(PRE(HEAD(name_s), TREE(tree_env)),
                         ENV(tree_env));
```

```
            end;
         else
            return TREE_ENV(MODULE_NAME_EXPECTED, env);
         end if;
      end USE_ENV;

      procedure USE_OP(env1, env2: S_ENV) return S_ENV is
         env1_part1: constant HALF_S_ENV := SURROUNDING(env1);
         env2_part1: constant HALF_S_ENV := SURROUNDING(env2);
         env1_part2: constant HALF_S_ENV := IMPORTED(env1);
         env_part2 : constant HALF_S_ENV := MERGE(env1_part2, env2_part1);
      begin
         return S_ENV(env1_part1, env_part2);
      end USE_OP;

      procedure MERGE(half_env1, half_env2: HALF_S_ENV) return HALF_S_ENV is
      begin
      -- <to be defined>
      end MERGE;
```

8.5 Renaming

A renaming declaration associates a local name with an entity.

Syntax:

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

Abstract Syntax:

```
renaming -> RENAME_SPEC  EXT_NAME   -- [RENAME_SPEC renames EXT_NAME; ]

RENAME_SPEC ::= subprogram | void | TYPE
EXT_NAME    ::= NAME | selected_string | string
```

Description:

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.

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.

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.

Syntax:

```
compilation ::= {compilation_unit}

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

Abstract Syntax:

```
restricted -> NAME_S   DECL_S      -- [ restricted NAME_S   DECL_S ]
separate   ->                      -- [ separate ]
```

Description:

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.

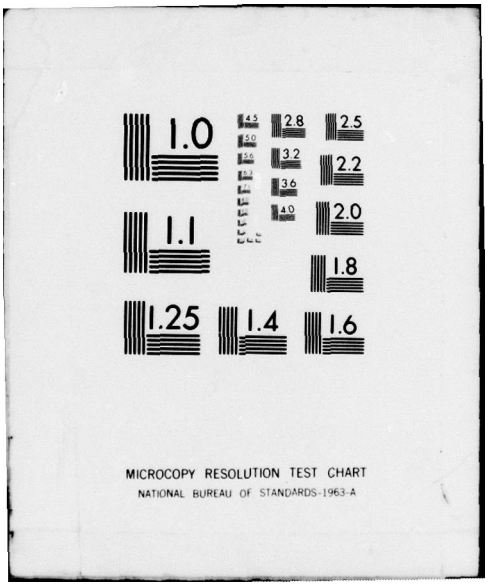
A compilation unit that is a subprogram body can be a main program in the usual sense. The means by which main programs are executed are not within the language definition.

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).

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.

AD-A073 714 HONEYWELL SYSTEMS AND RESEARCH CENTER MINNEAPOLIS MN F/G 9/2
THE GREEN LANGUAGE. A FORMAL DEFINITION.(U)
APR 79 MDA73-77-C-0331
UNCLASSIFIED NL

2 OF 2
AD
A073714


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

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.

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.

Syntax:

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

Abstract Syntax:

```
stub ->      -- [ is separate ]
```

Description:

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.

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.

Syntax:

```
exception_declaration ::= identifier_list : exception;
```

Abstract Syntax:

```
decl -> NATURE DESIGNATOR_S DESCRIPTION

NATURE      ::= constant    | entry     | exception | function         | in   | in out | out |
                package     | procedure | subtype   | task             | type | variable
DESCRIPTION ::= instantiation | object  | renaming  | type description | unit | void
```

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.

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.

Syntax:

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

exception_choice ::= exception_name | others
```

Abstract Syntax:

```
alternative -> CHOICE S STM S      -- [ when CHOICE S => STM S ]
choice s    -> CHOICE ...          -- [ CHOICE | ... ]
stm s       -> STM ...             -- [ STM ... ]
```

```
CHOICE_S ::= choice_s
CHOICE   ::= EXP | others | RANGE
STM_S    ::= stm_s
```

Description:

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.

Static Semantics:

```
procedure CHECK_HANDLER(alternative: TREE; env: S_ENV) return TREE is
   env2: S_ENV := IN_HANDLER(env);
begin
   return MAKE(alternative, CHECK_EX_CHOICE_S(alternative), env2),
                            CHECK_STM_S(STM_S(alternative), env2));
end;

procedure CHECK_EX_CHOICE_S(choice_s: TREE; env: S_ENV) return TREE is
begin
   if IS_EMPTY(choice_s) then return void;
   else
      return PRE(CHECK_EX_CHOICE  (HEAD(choice_s), env),
                 CHECK_EX_CHOICE_S(TAIL(choice_s), env));
   end if;
end CHECK_EX_CHOICE_S;

procedure CHECK_EX_CHOICE(choice: TREE; env: S_ENV) return TREE is
begin
   if IS_NAME(KIND(choice)) then
      return CHECK_EX_NAME(choice, env);
   elsif choice = others then
      return choice;
   else return EXCEPTION_CHOICE_MUST_BE_A_NAME;
   end if;
end CHECK_EX_CHOICE;

procedure CHECK_EX_NAME(name: TREE; env: S_ENV) return TREE is
begin
   if KIND(name) = selected and IS_TASK(DEN_OF(NAME(name), env)) then
      return CHECK_EX_TASK(name, env);
   else
```

```
        return ID_PART(DEN_OF(name, env));
      end if;
   end CHECK_EX_NAME;
```

11.3 Raise Statements

An exception can be explicitly raised by a raise statement.

Syntax:

```
raise_statement ::= raise [exception_name];
```

Abstract Syntax:

```
raise -> NAME_VOID  [raise NAME_VOID ]

NAME_VOID ::= NAME | void
```

Description:

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.

Static Semantics:

```
procedure CHECK_RAISE(raise: TREE; env: S_ENV) return TREE is
begin
   case KIND(NAME_VOID(raise)) of
      when void =>    --  [ raise; ]
         if IS_IN_HANDLER(env) then
            return raise;
         else return RAISE_ALONE_CAN_ONLY_APPEAR_IN_A_HANDLER;
         end if;
      when others =>  -- [raise NAME; ]
         return MAKE(raise, CHECK_EX_NAME(NAME_VOID(raise), env));
   end case;
end CHECK_RAISE;
```

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).

11.4 Exceptions Raised during Tasking

An exception can be propagated to a task communicating, or attempting to communicate, with another task.

On any attempt to call an entry or a subprogram of an inactive task, the TASKING_ERROR exception is raised in the invoking task. Note that this also applies to entry calls to an active task if the task terminates before accepting these calls.

A rendezvous can be terminated abnormally in three cases.

(a) When an exception is raised inside an accept statement and not handled locally. In this case, the exception is propagated both in the unit containing the accept statement and in the calling task at the point of the entry call. (A different treatment is employed for the exception FAILURE as explained in section 11.5 below.)

(b) When the unit containing the accept statement is terminated abnormally (e.g. as the result of an abort statement). In this case, the TASKING_ERROR exception is raised in the calling task at the point of the entry call.

(c) When the unit issuing the entry call is terminated abnormally. In this case the rendezvous terminates abnormally and the TASKING_ERROR exception is raised within the called task, in the unit containing the accept statement.

A task calling a procedure of another task receives a TASKING_ERROR exception if the called task terminates before the end of the procedure execution.

11.5 Raising an Exception in Another Task

A task can raise the predefined exception FAILURE in another task (say T) by a raise statement of the form:

```
raise T.FAILURE;
```

The execution of this statement has no direct effect on the task issuing the statement (unless, of course, it raises FAILURE for itself).

If the task receiving the FAILURE exception is currently executing, or if it is suspended by an accept or select statement, the effect is to raise the exception at the point of the current statement. If the task is suspended on a delay statement, the corresponding wait is cancelled and the exception is raised at the point of the delay statement. If the task has issued an entry call, the exception is raised at the point of the call and two cases are possible for the called task:

(a) If the entry call has not yet been accepted, the call is cancelled and the called task is unaffected.

(b) If an accept statement for this entry is in execution, the rendezvous is abnormally terminated and the TASKING_ERROR exception is raised, as in section 11.4(c).

If a FAILURE exception is received by a suspended task, execution of the task is scheduled according to the priority rules (see 9.8) in order to allow handling of the exception. If the exception FAILURE is received within an accept statement and not handled locally, the rendezvous is terminated and the exception TASKING_ERROR is raised in the calling task at the point of the entry call.

The predefined exception FAILURE is the only exception that can be explicitly raised in another task. It supersedes all other exceptions not yet handled or received before FAILURE is handled. A unit can contain a handler for the exception FAILURE as for any other exception.

11.6 uppressing Exceptions

The detection of exception conditions may be suppressed within a unit by a pragma of the form:

pragma SUPPRESS(exception_name {, exception_name})

This pragma indicates that no run time checks need be provided to ensure that the named exceptions do not arise. The occurrence of such a pragma within a given unit does not guarantee that the named exceptions will not arise since the pragma is merely a recommendation to the compiler, and since the exceptions may be propagated by called units. Should an exception situation occur when the corresponding run time checks are omitted, the program would be erroneous, and the results unpredictable.

A. APPENDIX: ABSTRACT SYNTAX OF GREEN

```
package GREEN_SYNTAX is
   type CONSTRUCT is (

      -- nullary
      and          , and then          , catenate     , constant       , div          ,
      entry        , eq                , exception    , exponentiation , function     ,
      ge           , gt                , id           , in             , in out       ,
      in op        , int number        , ic           , it             , minus        ,
      mult         , mod               , ne           , not            , not in       ,
      null         , others            , or           , or else        , out          ,
      package      , packing           , plus         , private        , procedure    ,
      real number  , restricted private , separate     , stub           , string       ,
      subtype      , task              , type         , variable       , void         ,

      -- unary
      abort        , access            , address      , all            , allocator    ,
      assert       , code              , delay        , derived        , goto         ,
      initiate     , raise             , return       , subunit        , type description ,
      while        , integer           ,

      -- binary
      alternative  , array             , assign       , call           , case         ,
      conditional  , constrained       , exit         , family         , fixed        ,
      float        , for               , generic      , is assoc       , if           ,
      in assoc     , indexed           , in out assoc , instantiation  , labeled      ,
      loop         , module            , named        , object         , out assoc    ,
      pair         , predefined        , qualified    , renaming       , reverse      ,
      restricted   , select            , selected     , selected string , slice       ,
      subprogram   , typed pair        , unary        , unit           , use          ,
      variant      , variant part      ,

      -- ternary
      accept       , binary            , block        , comp repr      , condition    ,
      decl         , membership        , record repr  ,

      -- arbitrary
      alternative s , bounds s         , choice s     ,
      comp s        , comp assoc s     , comp repr s  ,
      conditional s , decl s           , designator s ,
      exp s         , item s           , name s       ,
      param assoc s , range s          , variant s    )

   type ARITIES is (nullary, unary, binary, ternary, arbitrary);
   function ARITY (construct: CONSTRUCT) return ARITIES;
```

```
type SORT is set_of (CONSTRUCT);
   -- We assume a generic package set has been defined
   -- which provides sets and union of sets

                                        -- Table of all sorts

   ALTERNATIVE   , ALTERNATIVE_S , BINARY_OP    , BODY          , BOUNDS     ,
   BOUNDS_S      , CHOICE        , CHOICE_S     , COMP          , COMP_S     ,
   COMP_ASSOC    , COMP_REPR     , COMP_REPR_S  , COND          ,            ,
   COND_VOID     , CONDITION_OP  , CONDITIONAL  , CONDITIONAL_S ,            ,
   CONSTRAINT    , CONSTRAINED   , DECL         , DECL_S        , DECL_PART  ,
   DESCRIPTION   , DESIGNATOR    , DESIGNATOR_S , EXP           , EXP_VOID   ,
   EXT_NAME      , GENERIC_ASSOC , ID           , ID_VOID       , ITEM       , ITEM_S,
   ITERATION     , MEMBER_OP     , NAME         , NAME_S        , NAME_VOID,
   NATURE        , PAIR          , PARAM_ASSOC  , PARAM_ASSOC_S , QUALIFIED,
   RANGE         , RANGE_VOID    , RESULT       , SPECIFIER     , STM        ,
   STM_S         , STRING        , TYPE         , TYPE_RANGE    , TYPE_SPEC,
   UNARY_OP      , VARIANT       , VARIANT_S
                                                : constant SORT;

   function SORT_OF_SON(construct: CONSTRUCT; n: INTEGER := 0) return SORT;
      -- the expression SORT_OF_SON(construct, n) denotes the sort of the
      -- n-th argument of "construct", if it is of fixed arity.  In the case
      -- of a list construct, it denotes the common sort of each son.

private

   -- The sorts are initialized so as to be the sets described by the following table:

-- ALTERNATIVE   ::= alternative
-- ALTERNATIVE_S ::= alternative_s

-- BINARY_OP ::= and    | catenate | div  | eq         | exponentiation | ge  |
--               gt     | le       | lt   | minus      | mult           | mod |
--               ne     | not      | or   | plus
-- BODY      ::= block  | stub     | void | EXT_NAME
-- BOUNDS    ::= id     | indexed  | pair | predefined | selected       |
--               typed_pair
-- BOUNDS_S  ::= bounds_s

-- CHOICE    ::= EXP | others | RANGE
-- CHOICE_S  ::= choice_s
-- COMP      ::= decl           | null       | variant_part
-- COMP_S    ::= comp_s
-- COMP_ASSOC::= named          | all        | allocator  | binary | call | comp_assoc_s |
--               float_number   | id         | int_number | indexed      | membership |
--               null           | predefined | qualified  | real_number  | selected   |
--               selected_string| slice      | string     | unary
-- COMP_REPR  ::= comp_repr
-- COMP_REPR_S   ::= comp_repr_s
-- COND          ::= condition | EXP
-- COND_VOID     ::= COND | void
-- CONDITION_OP  ::= and_then| or_else
-- CONDITIONAL   ::= conditional
```

```
-- CONDITIONAL_S ::= conditional_s
-- CONSTRAINT    ::= fixed       | float    | comp_assoc_s | pair |
--                   typed_pair | range_s | void
-- CONSTRAINED   ::= constrained

-- DECL          ::= decl
-- DECL_S        ::= decl_s
-- DECL_PART     ::= use           | item_s
-- DESCRIPTION   ::= instantiation | object | renaming | type_description |
--                   unit          | void
-- DESIGNATOR    ::= id            | string
-- DESIGNATOR_S  ::= designator_s

-- EXP           ::= all           | allocator      | binary     | call      | comp_assoc_s |
--                   float_number  | id             | int_number | indexed   |
--                   membership    | null           | predefined | qualified |
--                   real_number   | selected       | slice      |
--                   string        | unary
-- EXP_VOID      ::= EXP           | void
-- EXT_NAME      ::= NAME          | selected_string | string

-- GENERIC_ASSOC ::= PARAM_ASSOC  | selected_string | is_assoc


-- ID            ::= id
-- ID_VOID       ::= id            | void
-- ITEM          ::= address       | decl     | comp_repr | packing    | record_repr |
--                   restricted    | subunit  | EXP
-- ITEM_S        ::= item_s
-- ITERATION     ::= for           | reverse  | void      | while
-- MEMBER_OP     ::= in_op         | not_in
-- NAME          ::= all           | id       | indexed   | predefined | selected    | slice
-- NAME_S        ::= name_s
-- NAME_VOID     ::= NAME          | void
-- NATURE        ::= constant      | entry    | exception | function   |
--                   in            | in_out   | out       | package    | procedure   | subtype |
--                   task          | type     | variable

-- PAIR          ::= pair
-- PARAM_ASSOC   ::= in_assoc | in_out_assoc | out_assoc | EXP
-- PARAM_ASSOC_S ::= param_assoc_s

-- QUALIFIED     ::= qualified

-- RANGE         ::= pair        | typed_pair
-- RANGE_VOID    ::= void | RANGE
-- RESULT        ::= constrained | void

-- SPECIFIER ::= family  | generic | module          | subprogram | void
-- STM       ::= accept  | abort   | assign | assert | block      | call | case     |
--               code    | delay   | exit           | goto       | if   | initiate |
--               labeled | loop    | null           | raise      | return | select
-- STM_S     ::= stm_s
-- STRING    ::= string
```

```
-- TYPE       ::= access      | array        | comp s              | constrained |
--                derived     | designator s | fixed               | float       |
--                integer     | private      | restricted private  | void
-- TYPE RANGE ::= constrained | pair         | typed pair
-- TYPE SPEC  ::= subprogram  | void | TYPE

-- UNARY OP ::= minus | not | plus

-- VARIANT   ::= variant
-- VARIANT S ::= variant s
```

```
-- The constructs of Green have a structure described
-- by the following tables:

-- Unary constructs

-- abort              -> NAME S
-- access             -> TYPE
-- address            -> EXP
-- all                -> NAME
-- allocator          -> QUALIFIED
-- assert             -> COND
-- code               -> QUALIFIED
-- delay              -> EXP
-- derived            -> CONSTRAINED
-- goto               -> ID
-- initiate           -> NAME S
-- integer            -> RANGE
-- raise              -> NAME VOID
-- return             -> EXP VOID
-- subunit            -> DECL
-- type description   -> TYPE
-- while              -> EXP

-- Binary constructs

-- alternative        -> CHOICE S       STM S
-- array              -> BOUNDS S       TYPE
-- assign             -> NAME           EXP
-- call               -> NAME           PARAM ASSOC S
-- case               -> EXP            ALTERNATIVE S
-- conditional        -> COND           STM S
-- constrained        -> NAME           CONSTRAINT
-- exit               -> ID VOID        COND VOID
-- family             -> RANGE          SPECIFIER
-- fixed              -> EXP            RANGE VOID
-- float              -> EXP            RANGE VOID
-- for                -> ID             RANGE
-- generic            -> DECL S         SPECIFIER
-- if                 -> CONDITIONAL S  STM S
-- in assoc           -> ID             EXP
-- indexed            -> NAME           EXP S
-- in out assoc       -> ID             EXP
-- instantiation      -> NAME           PARAM ASSOC S
-- is assoc           -> DESIGNATOR     EXT NAME
-- labeled            -> ID             STM
-- loop               -> ITERATION      STM S
-- module             -> DECL PART      DECL PART
-- named              -> CHOICE S       EXP
-- object             -> TYPE           EXP VOID
-- out assoc          -> ID             EXP
-- pair               -> EXP            EXP
-- predefined         -> NAME           ID
-- qualified          -> NAME           EXP
-- renaming           -> TYPE SPEC      EXT NAME
-- reverse            -> ID             RANGE
```

```
-- restricted        -> NAME_S          DECL_S
-- select            -> CONDITIONAL_S   STM_S
-- selected          -> NAME            ID
-- selected string   -> NAME            STRING
-- slice             -> NAME            RANGE
-- subprogram        -> DECL_S          RESULT
-- typed pair        -> NAME            PAIR
-- unary             -> UNARY_OP        EXP
-- unit              -> SPECIFIER       BODY
-- use               -> NAME_S          ITEM_S
-- variant           -> CHOICE_S        COMP_S
-- variant part      -> NAME            VARIANT_S

-- Ternary constructs

-- accept            -> NAME            DECL_S          STM_S
-- binary            -> EXP             BINARY_OP       EXP
-- block             -> DECL_PART       STM_S           ALTERNATIVE_S
-- comp repr         -> NAME            EXP             RANGE
-- condition         -> EXP             CONDITION_OP    COND
-- decl              -> NATURE          DESIGNATOR_S    DESCRIPTION
-- membership        -> EXP             MEMBER_OP       TYPE_RANGE
-- record repr       -> NAME            EXP             COMP_REPR_S

-- List constructs

-- alternative_s     -> ALTERNATIVE     ...
-- bounds_s          -> BOUNDS          ...
-- choice_s          -> CHOICE          ...
-- comp_s            -> COMP            ...
-- comp_assoc_s      -> COMP_ASSOC      ...
-- comp_repr_s       -> COMP_REPR       ...
-- conditional_s     -> CONDITIONAL     ...
-- decl_s            -> DECL            ...
-- designator_s      -> DESIGNATOR      ...
-- exp_s             -> EXP             ...
-- item_s            -> ITEM            ...
-- name_s            -> NAME            ...
-- param_assoc_s     -> PARAM_ASSOC     ...
-- range_s           -> RANGE           ...
-- variant_s         -> VARIANT         ...


end GREEN_SYNTAX;
```

```
package GREEN_TREES is
   use GREEN_SYNTAX;
   type TREE is private;
                -- Tree constructors

   procedure MAKE(construct: CONSTRUCT; s: STRING)         return TREE;
   procedure MAKE(construct: CONSTRUCT; t: TREE)           return TREE;
   procedure MAKE(construct: CONSTRUCT; t1, t2: TREE)      return TREE;
   procedure MAKE(construct: CONSTRUCT; t1, t2, t3: TREE) return TREE;

                -- Tree selectors

   procedure KIND    (t: TREE)                 return CONSTRUCT;
   procedure SON     (n: INTEGER, t: TREE)     return TREE;
   procedure TOKEN   (t: TREE)                 return STRING;

                -- Handling of list constructs

   procedure HEAD    (l: TREE)                 return TREE;
   procedure TAIL    (l: TREE)                 return TREE;
   procedure PRE     (t: TREE, l: TREE)        return TREE;
   procedure EMPTY   (construct: CONSTRUCT)    return TREE;
   procedure IS_EMPTY(l: TREE)                 return BOOLEAN;

private
   -- Description of the implementation of type TREE
end GREEN_TREES;
```

```
package GREEN_SELECTORS is
   use GREEN_SYNTAX, GREEN_TREES;
   -- To increase readability, one wishes to avoid positional selection of subtrees.
   -- To this effect, selector functions are defined that are named after the sort
   -- of the corresponding subtree.
   -- Example:  if: constant TREE -- where KIND(if) = if
   -- then SON(1, if) is equivalent to CONDITION_S(if) and
   --      SON(2, if) is equivalent to STM_S(if)
   -- Hence the following selectors are declared:
      procedure CONDITION_S(t: TREE) return TREE;
      procedure STM_S(t: TREE) return TREE;
   -- and so on for each construct of fixed arity.  In cases like pair
   -- which has more than one son of the same sort, numbering is used;
      procedure EXP1(t: TREE) return TREE;
      procedure EXP2(t: TREE) return TREE;
end GREEN_SELECTORS;
```