The Programming Language Declarative Ada Reference Manual

John Thornley
Computer Science Department
California Institute of Technology
Pasadena, California 91125, USA
john-t@cs.caltech.edu

April 24, 1993

Contents

1 Introduction

Declarative Ada is a parallel declarative programming language based on a subset of the programming language Ada [1].

This manual defines the syntax of Declarative Ada in extended BNF and describes the semantics in English. Wherever possible, the same notation and terminology is used as in the Ada Reference Manual [1]. Unless otherwise stated, constructs in Declarative Ada have the same meaning as they do in Ada. Therefore, the Ada Reference Manual provides a more detailed description of most Declarative Ada constructs.

This manual is not intended to be a tutorial: a discussion of parallel programming with Declarative Ada is contained in [4]; a collection of Declarative Ada example programs are described in [3]; and an excellent introduction to Ada is given by [2].

1.1 Language Outline

Declarative Ada is based on a sufficiently powerful subset of Ada to demonstrate that declarative parallel programs can be written with a modified version of Ada. Many of the features that make Ada a useful language for large scale software engineering are omitted from Declarative Ada to keep the language small.

Program Structure and Declarations

Programs consist of a sequence of constant, type, and subprogram (procedure and function) declarations. One subprogram is executed as the *main program*. Parameters of procedures can be of **in**, **out**, and **in out** mode. Parameters of functions must be of **in** mode. Subprograms can contain local variable declarations. There are no global variable declarations, local constant or type declarations, or nested subprogram declarations. There are no packages, tasks, exception handling, or generic units.

Statements

Statements can be of the following kinds: assignment, block, if-then-else, for-loop, null, procedure-call, and function return. The Ada *sequence of* *statements* is replaced by the *composition of statements*, which can be specified as **parallel** or **sequential**. Similarly, the iterations of loop statements can be specified as **parallel** or **sequential**. There are no case, while-loop, exit, or goto statements, and no statements that relate to tasks or exception handling.

Data Types

Integer, floating-point, Boolean, and string types are predefined. Array, record, and access types can be defined. There are no character, enumeration, range, fixed-point, unconstrained array, or variant record types. There are no subtypes or derived types.

Operators

Logical, relational, and arithmetic operators from Ada can be used. There are strict logical operators, where both operands are always evaluated, and short-circuit forms, where the right operand is only evaluated if necessary. Components can be indexed from arrays and selected from records, and array and record values can be constructed from aggregates of components. Designated objects can be allocated using the **new** operator, with deallocation being the responsibility of the implementation.

1.2 Parallelism

1.2.1 The Single-Assignment Restriction

All variables are *single-assignment variables*, subject to the following restrictions:

- Variables are initialized to a special *undefined* value.
- Evaluation of an undefined variable as an expression suspends until the variable is assigned a value.
- A variable can be assigned a value at most once.

The single-assignment restriction changes Declarative Ada from a sequential imperative subset of Ada to a parallel declarative programming language.

1.2.2 Parallel Processes

A program is implicitly executed as a group of *processes*, communicating and synchronizing though shared single-assignment variables. A process is any of the following units of computation:

- The execution of a statement.
- The execution of a composition of statements.
- The evaluation of a variable name.
- The evaluation of an expression.

Processes are *executed in parallel* in the sense that the result of executing a program is equivalent to the result of a fair interleaving of the atomic actions of the individual processes. A program can be considered to be executed on an arbitrarily large number of processors, with each process executed on a separate processor. The result of executing a program is independent of whether processes are actually executed: truly concurrently on separate processors, interleaved on a single processor, in some acceptable sequential order on a single processor, or a combination of the above. (The only execution requirement is that the fairness rule given in Section 1.2.5 is satisfied.)

Note: The evaluation of a name on the left-hand side of an assignment statement, or as an **out** or **in out** mode actual parameter is the evaluation of the name as a *variable name.* The evaluation of a name on the right-hand side of an assignment statement, or as an **in** mode actual parameter is the evaluation of the name as an *expression.*

1.2.3 Process Initiation and Termination

Any non-trivial process implicitly *initiates* other parallel processes, through the execution of enclosed statements and compositions of statements, and the evaluation of enclosed variable names and expressions. A process *terminates* after it has no further actions to perform and all of its subprocesses have terminated. This single mechanism explains parallel execution at all levels of granularity. In this manner, parallelism is implicit in the execution of the following language constructs:

1. **Statements:** The executions of enclosed compositions of statements and the evaluations of enclosed variable names and expressions are parallel processes. For example, unless the loop is specified as **sequential**, the iterations of a loop statement are executed in parallel.

2. **Compositions of Statements:** Unless the composition is specified as **sequential**, the executions of the enclosed statements are parallel processes. For example, procedure calls enclosed in a parallel composition of statements can be thought of as conventional coarse-grain parallel processes.

3. **Variable Names and Expressions:** The evaluations of enclosed prefixes and subexpressions are parallel processes. For example, the actual parameters of a function call are evaluated in parallel with each other and the execution of the function body. The actual parameters can themselves be function calls, giving parallelism from functional composition.

1.2.4 Executable and Suspended Processes

Between its initiation and termination, every process is either *executable* or *suspended*. A process becomes suspended when either:

1. The process is the evaluation of a name as an expression, and the value of the name is undefined. In this case, the process becomes executable after the name is assigned a value by another parallel process.

2. To perform any further actions, the process requires values that have not yet been evaluated by its subprocesses. In this case, the process becomes executable after its subprocesses have evaluated the required values.

3. The process has no further actions to perform, but some of its subprocesses have not yet terminated. In this case, the process terminates after all of its subprocesses have terminated.

An executable process remains executable until it is executed. It is impossible for an executing process to change another process from executable to suspended or terminated.

1.2.5 Fairness

An implementation of the language can schedule processes in any manner such that the following *fairness rule* is satisfied:

> *From any point in time in a program's execution, every executable process will eventually have some progress made on its execution.*

The fairness rule is required to prevent infinite processes from "locking out" other processes. This is a weak form of fairness—there is no requirement to share processing time in any sense "evenly" amongst processes. In a finite program execution with no feedback in data flow, a sequential ordering of processes can always be found that satisfies the fairness rule.

1.2.6 Communication and Synchronization

Communication and synchronization between parallel processes is implicit: communication is through shared variables, and synchronization is through one process suspending until another process assigns a value to a shared variable. This single mechanism explains communication and synchronization between parallel processes at all levels of granularity. In this manner, the scheduling of processes is automatically constrained by the data flow that occurs during the execution of a program.

For example, in *producer-consumer* interactions, a *producer* process generates as output a data-structure that is used as input by a *consumer* process. The consumer suspends whenever it attempts to use the value of an undefined component of the data-structure, and it resumes execution after the producer assigns a value to that component. It is not required that all the components of the shared data-structure be assigned values by the producer before the consumer is executed.

1.3 Syntax Notation

The syntax of the language is described using a simple variant of BNF:

(a) Lower case words, some containing embedded underlines, are used to denote syntactic categories. For example:

adding_operator

(b) Boldface words are used to denote reserved words. For example:

array

(c) Square brackets enclose optional items. For example:

parameter_association ::= [mode] actual_parameter

(d) Braces enclose a repeated item that can appear zero or more times. For example:

term ::= factor {multiplying_operator factor}

(e) A vertical bar separates alternative items. For example:

mode ::= **in** | **out** | **in out**

(f) If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *type*_name is equivalent to name alone.

1.4 Execution Errors

Throughout the language definition, the term *error* is used to describe execution error conditions that cannot, in general, be checked at compile time. The result of a program with an execution error is implementation-dependent and not necessarily deterministic.

2 Lexical Elements

2.1 Character Set

The only characters allowed in the text of a program are *graphic characters* and *format effectors*.

The graphic characters consist of:

(a) 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

(b) 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

(c) digits
0 1 2 3 4 5 6 7 8 9

(d) special characters
" # & ' ( ) * + , − . / : ; < = > _ | ! $ % ? @ [\ ] ^ ` { } ~

(e) the space character

The format effectors consist of horizontal tabulation, vertical tabulation, carriage return, line feed, and form feed.

2.2 Lexical Elements and Separators

Characters in a program are composed into *lexical elements* of the following kinds:

(a) Delimiters

(b) Identifiers

(c) Numeric literals

(d) Character literals

(e) String literals

(f) Comments

(g) Reserved words

One or more *separators* are allowed between any two adjacent lexical elements, at the start of the program, or at the end of the program. The following are separators:

(a) The space character, except within a comment, string literal, or space character literal.

(b) The end of line, which is signified by any format effector other than horizontal tabulation.

(c) Horizontal tabulation, except within a comment.

Note:

Some of the defined lexical elements do not occur anywhere in the syntax of Declarative Ada. They are included solely for compatibility with Ada.

2.3 Delimiters

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

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

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

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

Delimiters have special significance in the language.

2.4 Identifiers

identifier ::= letter {[underline] letter_or_digit}

letter_or_digit ::= letter | digit

letter ::= upper_case_letter | lower_case_letter

Identifiers are used as names. All characters of an identifier are significant. Identifiers differing only in the use of corresponding upper and lower case letters are considered the same. No identifier can have the same spelling as a reserved word.

Examples:

COUNT	X	get_symbol	Ethelyn	Marion
SNOBOL_4	X1	PageCount	STORE_NEXT_ITEM	

2.5 Numeric Literals

numeric_literal ::= integer [.integer] [exponent]

integer ::= digit {[underline] digit}

exponent ::= E [+] integer | E − integer

There are two classes of numeric literals: real literals and integer literals. A real literal is a numeric literal that includes a point. An integer literal is a numeric literal without a point.

An underline character within a numeric literal does not affect its value. The letter E of the exponent can be written either in lower case or in upper case. The exponent for an integer literal must not have a minus sign.

Examples:

12	0	1E6	123_456	−− integer literals
12.0	0.0	0.456	3.14159_26	−− real literals
1.34E−12	1.0E+6	−− real literals with exponent		

2.6 Character Literals

character_literal ::= 'graphic_character'

A character literal consists of a graphic character between two apostrophe characters.

Examples:

'A'   '*'   '''   ' '

2.7 String Literals

string_literal ::= "{graphic_character}"

A string literal consists of a (possibly empty) sequence of graphic characters enclosed between two quotation characters. If a quotation character is to be represented in the sequence of characters, then a pair of adjacent quotation characters must be written at the corresponding place within the string literal.

Examples:

"Message of the day:"

""   −− an empty string literal
" "   "A"   """"   −− three string literals of length 1

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

2.8 Comments

A comment starts with two adjacent hyphens and extends up to the end of the line. A comment has no influence on the meaning of a program.

Examples:

end; −− processing of LINE is complete

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

−−−−−−−−−−−−−−−−− the first two hyphens start the comment

2.9 Reserved Words

The words listed below are called *reserved words* and have special significance in the language.

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

A reserved word must not be used as an identifier. Reserved words differing only in the use of corresponding upper and lower case letters are considered the same.

Note:

The reserved words consist of all the Ada reserved words and the additional words **parallel** and **sequential**.

3 Identifiers, Scope, and Visibility

3.1 Named Entities

Identifiers can be declared to name the following entities:

- Constant objects.
- Variable objects.
- Types.
- Components of record types.
- Procedure and function subprograms.
- Formal parameters of subprograms.
- Loop parameters.

3.2 Declarative Regions

Each declaration is *local* to a *declarative region*. A given identifier cannot be declared more than once in the local declarations of a declarative region.

A declarative region is formed by the text of each of the following:

- A program.
- A subprogram declaration.
- A record type declaration.
- A loop statement.

Declarative regions can be nested within other declarative regions.

3.3 Scope and Visibility

The *scope* of a declaration extends from the beginning of the declaration to the end of the declarative region. The *visibility* of a declaration defines where in the scope of a declaration it is legal to use the identifier to refer to the declared entity.

Unless stated otherwise, a declaration is visible throughout its scope except in those places where it is *hidden* by the scope of a declaration of the same identifier within an enclosed declarative region. Additionally, a component declaration has scope and visibility in the selector of a selected component whose prefix is the appropriate record or access type.

4 Constant Declarations

constant_declaration ::=
    identifier : **constant** type_mark := expression;

A constant declaration declares an object with a value defined by an expression. The expression must be of the same type as the type mark. The value of a constant cannot be redefined.

The visibility of a constant declaration begins immediately after the declaration, i.e., self-referencing constants cannot be declared.

Examples:

rows   : **constant** integer   := 100;
columns : **constant** integer   := 2 * (rows + 1);
large   : **constant** Boolean := rows * columns > 1000;

5 Type Declarations

type_declaration ::=
    incomplete_type_declaration | full_type_declaration

incomplete_type_declaration ::=
    type identifier;

full_type_declaration ::=
    type identifier **is** type_definition;

type_definition ::=
    array_type_definition | record_type_definition
  | access_type_definition

type_mark ::= *type*_simple_name

A type declaration declares a type. Each type is a distinct type.

An incomplete type declaration declares a type for which a full type declaration must occur later. Until the full type declaration occurs, an incomplete type can only be used as the designated type in an access type definition.

The visibility of a type declaration begins immediately after the declaration, i.e., self-referencing types cannot be declared.

Examples:

```
type node;
type list   is access node;
type node   is record
                  head : integer;
                  tail : list;
               end record;
type vector is array(0 .. N−1) of float;
```

5.1 Predefined Scalar Types

5.1.1 Integer Type

An integer object is a numeric object that can be defined to have an integer value in an implementation-dependent range.

5.1.2 Float Type

A float object is a numeric object that can be defined to have a floating-point value in an implementation-dependent range with implementation-dependent precision.

5.1.3 Boolean Type

A Boolean object is a enumeration object that can be defined to have the value of one of the two predefined Boolean constants: false and true. For ordering purposes, false precedes true.

5.2 Predefined String Type

A string object is an object that can be defined to have the value of a string literal. The only significant use for strings is in output operations.

5.3 Array Types

array_type_definition ::=
    array(range {, range}) **of** type_mark

range ::= *integer*_simple_expression .. *integer*_simple_expression

An array object is a *composite* object consisting of components that have the same type.

A *null* range is a range for which the lower bound is greater than the upper bound. If any of the index ranges is a null range, the array type has no components and all arrays of the type are *null* arrays.

Examples:

array(0 .. 3) **of** float
array(0 .. N−1, 0 .. N−1) **of** Boolean

5.4 Record Types

record_type_definition ::=
  record
    component_list
  end record

component_list ::=
  component_declaration {component_declaration}
 | **null**;

component_declaration ::=
  identifier {, identifier} : type_mark;

A record object is a *composite* object having named components of possibly differing types. If the component list is the word **null**, the record type has no components and all records of the type are *null* records.

Examples:

```
record
    null;
end record

record
    value       : integer;
    left, right : tree;
end record
```

5.5 Access Types

access_type_definition ::= **access** type_mark

Access to an object created by an allocator is achieved through an *access value* returned by the allocator. The access value is said to *designate* the object. The type of objects designated by an access type is called the *designated type* of the access type.

For each access type, there is a literal **null** that designates no object at all.

Example:

```
access node
```

6 Variable Declarations

variable_declaration ::=
    identifier {, identifier} : type_mark;

A variable initially has an undefined value. For a composite type, this means a value in which all components have undefined values. The value of a variable or component of a variable can be defined by an assignment statement.

Examples:

```
sum      : integer;
u, v     : vector;
unsorted : list;
```

7 Subprogram Declarations

subprogram_declaration ::=
    incomplete_subprogram_declaration | full_subprogram_declaration

incomplete_subprogram_declaration ::=
    subprogram_specification;

full_subprogram_declaration ::=
    subprogram_specification **is**
        {variable_declaration}
    begin
        composition_of_statements
    end *subprogram*_simple_name;

subprogram_specification ::=
    procedure identifier [formal_part]
  | **function** identifier [formal_part] **return** type_mark

formal_part ::=
    (parameter_specification {; parameter_specification})

parameter_specification ::=
    identifier {, identifier} : [mode] type_mark

mode ::= **in** | **in out** | **out**

A subprogram declaration declares a procedure or function. An incomplete subprogram declaration declares a subprogram for which a full subprogram declaration must occur later. The subprogram specification of the corresponding full subprogram declaration must be formed by exactly the same sequence of lexical elements (ignoring comments).

A formal parameter of a subprogram has one of the three following modes:

- **in**: The formal parameter can be read but not written.
- **out**: The formal parameter can be written but not read.
- **in out**: The formal parameter can be read and written.

The default formal parameter mode is **in**. All formal parameters of a function must be of mode **in**.

The body of a function subprogram must include one or more return statements specifying the returned value.

Examples:

```
function min(x, y : integer) return integer;

procedure find_min(a, b, c, d : in integer; result : out integer);

function min(x, y : integer) return integer is
begin
    if x < y then return x; else return y; end if;
end min;

procedure find_min(a, b, c, d : in integer; result : out integer) is
    temp1, temp2 : integer;
begin
    temp1 := min(a, b);
    temp2 := min(c, d);
    result := min(temp1, temp2);
end find_min;
```

8 Names

name ::= simple_name | indexed_component | selected_component

simple_name ::= identifier

prefix ::= name | function_call

A simple name denotes the entity associated with an identifier by its declaration.

A prefix occurs as part of an indexed_component or selected_component.

A function call cannot occur in the prefix of a name evaluated as a variable.

8.1 Evaluation

A name can be evaluated as a variable (e.g., on the left-hand side of an assignment, or as an **out** or **in out** mode actual parameter) or as an expression (e.g., on the right-hand side of an assignment, or as an **in** mode actual parameter).

Evaluation of a name as an expression suspends when the named object has an undefined value and terminates after the named object has a defined value.

Evaluation of a name as a variable does not suspend when the named object has an undefined value.

8.2 Simple Names

In the case of a name evaluated as an expression, a simple name can be a variable, constant, formal parameter of mode **in** or **in out**, or a loop parameter. In the case of a name evaluated as a variable, a simple name can be a variable or formal parameter of mode **out** or **in out**.

8.3 Indexed Components

indexed_component ::=
    prefix(*integer*_simple_expression {, *integer*_simple_expression})

An indexed component denotes a component of an array. The prefix must be an array or access an array. The expressions specify the index values for the component. There must be one such expression for each index position in the array type.

It is an error if the prefix value is **null**, or if any of the index expressions lie outside of the corresponding index range.

Examples:

data(i)
board(row, column)
paths(k)(i, j)
heap.**all**(i)
add(u, v)(i)

8.4 Selected Components

selected_component ::= prefix.selector

selector ::= *component*_simple_name | **all**

A selected component with a component simple name as a selector denotes a component of a record. The prefix must be a record or access a record.

A selected component with **all** as a selector denotes the object designated by an access value. The prefix must be an access type.

It is an error if the prefix value is **null**.

Examples:

sorted.head
sorted.tail.head
sorted.**all**
sorted.**all**.head
add(u, v).x

9 Expressions

expression ::=
    relation {**and** relation} | relation {**and then** relation}
  | relation {**or** relation} | relation {**or else** relation}
  | relation {**xor** relation}

relation ::=
    simple_expression [relational_operator simple_expression]

simple_expression ::=
    [unary_adding_operator] term {binary_adding_operator term}

term ::= factor {multiplying_operator factor}

factor ::= primary | **abs** primary | **not** primary

primary ::=
    numeric_literal | **null** | string_literal | *expression*_name | allocator
  | function_call | type_conversion | qualified_aggregate | (expression)

Examples:

count
not found
height * width / 2.0
(left + right) / 2
− deduction + price + tax
height > width
(cold **and** sunny) **or** warm   −− parentheses are required

Evaluation of an expression evaluates zero or more subexpressions and yields a result.

9.1 Operators

logical_operator	::= **and** | **and then** | **or** | **or else** | **xor**
relational_operator	::= = | /= | < | <= | > | >=
binary_adding_operator	::= + | −
unary_adding_operator	::= + | −
multiplying_operator	::= * | / | **mod** | **rem**
highest_precedence_operator	::= **abs** | **not**

The operators above are given in order of increasing precedence. For a sequence of operators of the same precedence level, the operators are associated with their operands in textual order from left to right.

Unless stated otherwise, all operators have their conventional meaning.

It is an error if the result of a numeric operation lies outside of the range that can be represented by the result type.

9.1.1 Logical Operators and Short-circuit Control Forms

The logical operators take two Boolean operands and yield a Boolean result. The operators **and**, **or**, and **xor** always evaluate both of their operands. The short-circuit forms **and then** and **or else** always evaluate their left operand first. If the left operand of **and then** is false, the right operand is not evaluated and the expression yields false. If the left operand of **or else** is true, the right operand is not evaluated and the expression yields true.

9.1.2 Relational Operators

The equality and inequality operators evaluate two operands of the same scalar or access type and yield a Boolean result. If the operands are of an access type, one of the operands must be **null**.

The ordering operators evaluate two operands of the same scalar type and yield a Boolean result.

9.1.3 Binary Adding Operators

The binary adding operators evaluate two operands of the same numeric type and yield a result of that type.

9.1.4 Unary Adding Operators

The unary adding operators evaluate a single numeric operand and yield a result of the same type.

9.1.5 Multiplying Operators

The multiplication and division operators evaluate two operands of the same numeric type and yield a result of that type.

The **mod** and **rem** operators evaluate two integer operands and yield an integer result.

It is an error if the value of the right operand of a division, **mod**, or **rem** operator is zero.

9.1.6 Highest Precedence Operators

The **abs** operator evaluates a single numeric operand and yields a result of the same type.

The **not** operator evaluates a single Boolean operand and yields a Boolean result.

9.2 Allocators

allocator ::=
  new type_mark | **new** qualified_aggregate

Evaluation of an allocator creates an object and yields an access value that designates the object. For an allocator with a qualified aggregate, the value of the designated object is given by the qualified aggregate. For an allocator without a qualified aggregate, the designated object has an undefined value.

The type of the access value returned by an allocator is determined from the context in which it occurs.

Examples:

new node
new node'(items.head, insert(item, items.tail))
new matrix'((1, 0, 0), (0, 1, 0), (0, 0, 1))

9.3 Function Calls

function_call ::=
  *function*_simple_name [actual_parameter_part]

Evaluation of a function call evaluates the actual parameter variables and expressions, and executes the function body. The value of a function call is defined by the execution of a return statement within the function body.

It is an error if no return statement is executed within the function body.

Example:

insert(unsorted.head, sort(unsorted.tail))

9.4 Type Conversions

type_conversion ::= *numeric*_type_mark(expression)

Evaluation of a type conversion evaluates the expression given as an operand and converts the resulting value to a specified *target* type. Type conversions are only allowed between numeric types. The conversion of a float value to an integer type rounds to the nearest integer. If the operand is halfway between two integers, rounding can be either up or down.

It is an error to convert a float value to an integer if the result is outside of the range that can be represented by an integer value.

Examples:

float(i)
integer(x)

9.5 Qualified Aggregates

qualified_aggregate ::= *composite*_type_mark'aggregate

aggregate ::= (component_association {, component_association})

component_association ::= expression | aggregate

Evaluation of a qualified aggregate evaluates component values and combines them into a composite value of an array or record type. The aggregate must have one component association for every component of the type. In the case of a record or one-dimensional array, each component association must be an expression of the same type as the corresponding component. In the case of a multi-dimensional array, each component association must be an aggregate.

Examples:

node'(item, **null**)
vector'(x, y, z)
matrix'((1, 0, 0), (0, 1, 0), (0, 0, 1))

10 Statements

```
statement ::=
    simple_statement | compound_statement

simple_statement ::=
    assignment_statement         |  null_statement
  | procedure_call_statement     |  return_statement

compound_statement ::=
    block_statement              |  if_statement
  | loop_statement
```

A statement is either simple or compound. A simple statement encloses no other statements. A compound statement can enclose compositions of other statements.

10.1 Compositions of Statements

composition_of_statements ::= [composition] statement {statement}

composition ::= **parallel** | **sequential**

Execution of a **parallel** composition consists of the parallel execution of the enclosed statements. Execution of a **sequential** composition consists of the execution of the enclosed statements in succession (execution of each statement terminates before execution of the next statement is initiated). The default composition is **parallel**.

10.2 Assignment Statements

assignment_statement ::= *variable*_name := expression;

Execution of an assignment statement consists of the evaluation of the variable name and expression, and the definition of the value of the variable to be the value of the expression.

The variable and expression must be of the same type.

In the case of array and record types, an assignment statement is the same as a parallel composition enclosing component-by-component assignment statements.

It is an error if the same variable or component of a variable is assigned a value more than once.

Examples:

```
paths(0) := edges;
paths(k)(i, j) := min(paths(k−1)(i, j),
                      paths(k−1)(i, k) + paths(k−1)(k, j));
left := new node'(items.head, left_tail);
```

10.3 Block Statements

```
block_statement ::=
    begin
      composition_of_statements
    end;
```

Execution of a block statement consists of the execution of the enclosed composition of statements.

Example:

```
begin
    sequential
    p(in a, out b);
    q(in b, out c);
    begin
        parallel
        r(in c, out d);
        s(in c, out e);
    end;
end;
```

10.4 If Statements

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

Execution of an if statement consists of the evaluation of one or more of the conditions, determining the execution of one or none of the enclosed compositions of statements.

Execution of an if statement begins by evaluating the first condition and proceeds as follows:

- If a condition evaluates to true, the corresponding composition of statements is executed.
- If a condition evaluates to false and it is not the final condition, the

next condition is evaluated.

- If the final condition evaluates to false and the if statement has an else part, the final composition of statements is executed.
- If the final condition evaluates to false and the if statement has no else part, execution of the if statement has no effect.

Example:

if left = **null then**
  result := right;
elsif right = **null then**
  result := left;
elsif left.head < right.head **then**
  merge(**in** left.tail, **in** right, **out** tail);
  result := **new** node'(left.head, tail);
else
  merge(**in** left, **in** right.tail, **out** tail);
  result := **new** node'(right.head, tail);
end if;

10.5 Loop Statements

loop_statement ::=
  for identifier **in** [**reverse**] range [composition] **loop**
    composition_of_statements
  end loop;

Execution of a loop statement consists of the evaluation of the range bounds, and the execution of the enclosed composition of statements zero or more times.

The identifier is the declaration of an integer *loop parameter*. For each execution of the composition of statements, the loop parameter is a constant defined to be a value from the loop range. The visibility of the loop parameter begins immediately after the range, i.e., the loop parameter value cannot be used in the range expressions.

If the lower bound is greater than the upper bound, execution of the loop statement has no effect.

If the lower bound is less than or equal to the upper bound, the enclosed composition of statements is executed once for each value of the range. If the loop composition is **parallel**, the executions occur in parallel. If the loop composition is **sequential**, the executions occur in succession (each execution terminates before the next execution is initiated). In the **sequential** case, the loop parameter values occur in increasing order, unless the word **reverse** is present, in which case the loop parameter values occur in decreasing order. The default loop composition is **parallel**.

Example:

```
for k in 1 .. N sequential loop
    for i in 1 .. N loop
        for j in 1 .. N loop
            paths(k)(i, j) := min(paths(k−1)(i, j),
                                  paths(k−1)(i, k) + paths(k−1)(k, j));
        end loop;
    end loop;
end loop;
```

10.6 Null Statements

null_statement ::= **null**;

Execution of a null statement has no effect.

10.7 Procedure Call Statements

procedure_call_statement ::=
*procedure*_simple_name [actual_parameter_part];

actual_parameter_part ::=
(parameter_association {, parameter_association})

parameter_association ::=
[mode] actual_parameter

actual_parameter ::=
expression | *variable*_name

Execution of a procedure call statement consists of the evaluation of the actual parameter variables and expressions, and execution of the procedure body.

In a procedure or function call:

- One actual parameter must occur for each formal parameter declared in the subprogram declaration.
- An actual parameter corresponding to a formal parameter of mode **in** must be an expression. In the execution of the subprogram body, the formal parameter name will be defined to have the value of the actual parameter expression.
- An actual parameter corresponding to a formal parameter of mode **out** or **in out** must be a variable name. In the execution of the procedure body, operations on the formal parameter name will be performed on the actual parameter variable.
- For the mode **in out**, the variable must not be a formal parameter of mode **out** or a component thereof.
- An actual parameter must be of the same type as the corresponding formal parameter.

- If a mode is associated with an actual parameter, it must be the same as the mode of the corresponding formal parameter.

Examples:

main;
partition(**in** pivot, **in** items, **out** left, **out** right);
partition(pivot, items.tail, left_tail, right);

10.8 Return Statements

return_statement ::= **return** expression;

Execution of a return statement consists of defining the result returned by a function. The type of the expression must be the same as the type returned by the enclosing function. A return statement cannot occur within a procedure.

If a return statement occurs within a sequential composition of statements, it must be the final statement in that composition. Similarly, if a compound statement that anywhere contains a return statement occurs within a sequential composition of statements, it must be the final statement in that composition.

A return statement cannot occur within a loop statement.

It is an error if more than one return statement is executed in the evaluation of a function call.

Example:

return merge(sort(split(list).left), sort(split(list).right));

11 Program Structure

program ::=
  {constant_declaration}
  {type_declaration}
  {subprogram_declaration | subprogram_body}

A program consists of a sequence of constant declarations, followed by a sequence of type declarations, followed by a sequence of subprogram declarations. The means by which one of the subprograms is initiated as the *main program* when the program is executed is not specified by the language definition.

12 Input and Output

Input and output operations are through calls to the predefined procedures get, put, and new_line, described in Section 13. To avoid nondeterminacy, the parallel execution of input and output statements is restricted in the following manner:

- It is an error if more than one parallel subprocess of a process executes an output statement.
- It is an error if more than one parallel subprocess of a process executes an input statement.

Deterministic sequential execution of input and output statements can be specified using **sequential** compositions of statements and **sequential** loop statements.

13 Predefined Environment

The following declarations are predefined in a declarative region that encloses the program:

```
type integer is predefined integer type;

type float is predefined float type;

type Boolean is (false, true);

type string is predefined string type;

procedure get(item : out item_type);
-- Reads item from the standard input file.
-- Item can be integer or float.

procedure put(item : in item_type);
-- Writes item to the standard output file.
-- Item can be integer, float, or string.

procedure new_line;
-- Writes an end of line to the standard output file.
```

14 Acknowledgment

The design of the programming language Declarative Ada was supported in part by Air Force Office of Scientific Research grant ASFOR-91-0070.

References

[1] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual.* ANSI/MIL-STD-1815A-1983. Springer Verlag, Berlin, 1983.

[2] Grady Booch. *Software Engineering with Ada.* Benjamin/Cummings, Menlo Park, California, 1983.

[3] John Thornley. *A Collection of Declarative Ada Example Programs.* CS-TR-93-05. Computer Science Department, California Institute of Technology, 1993.

[4] John Thornley. *Parallel Programming with Declarative Ada.* CS-TR-93-03. Computer Science Department, California Institute of Technology, 1993.