79-0621

LEVEL II

PRELIMINARY REFERENCE MANUAL FOR THE GREEN PROGRAMMING LANGUAGE

AD A073661

A language designed
in accordance with the
Ironman requirements

MDA 903-77-C-0331

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

and

Cii Honeywell Bull
68 Route de Versailles
78430 Louveciennes, France

April 15, 1978

1. INTRODUCTION

This report describes the Green language. Designed in accordance with the Ironman requirements of the Department of Defense, the Green language represents a new attempt to combine classical language features with features often found only in specialized languages. These include facilities for handling exceptional conditions, parallel processing, representation specifications for data, encapsulated definitions, low level input-output, and access to system dependent parameters.

1.1 Design Goals

The Green language was designed with three overriding concerns: a recognition of the importance of program reliability and maintenance, a deep concern for programming as a human activity, and efficiency.

The need for languages that promote reliability and maintenance is well established. Hence emphasis was placed on program readability over program writability. For example, the Green language requires that program variables be explicitly declared and that their type be specified. Automatic type conversion is generally prohibited. As a result, translators can insure that the types of objects satisfy their intended use. Furthermore, error prone notations have been avoided, and the language syntax avoids the use of encoded forms in favor of more English-like constructs. Finally, the language offers strong support for separate compilation of program units.

The concern for the human programmer was also stressed during the design. Above all, an attempt was made to keep the language as small as possible, avoiding special cases and elaborate features that often hinder rather than assist programming. The structure of the language minimizes the number of underlying concepts, and an attempt was made to integrate all features in a consistent and simple way. The fact that the form or meaning of a proposed construct was difficult to express in a systematic way was grounds for rejection of the construct.

No language can avoid the problem of efficiency. Languages that require overly elaborate translators or that lead to the inefficient use of storage or execution time force these inefficiencies on all machines and on all programs. Every construct in the Green language was examined in the light of present implementation techniques. Any proposed construct whose implementation was unclear or required excessive machine resources was rejected.

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

1.2 Language Summary

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

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

A procedure subprogram is the logical counterpart to a series of actions: for example, it may read in data, update variables, or produce some output. A function subprogram is the logical counterpart to a mathematical function for computing a value; unlike a procedure, a function can have no side effects. An exception subprogram is the logical means for handling a special situation that can occur dynamically during program execution, e.g. an arithmetic overflow, an invalid assertion, or a user defined exception situation.

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

A path is the basic unit for defining concurrent computations. Paths may be implemented on multiple processors or with interleaved execution on a single processor. Communication between paths is handled by associating boxes with each path. The boxes allow for synchronization between paths and for transmission of data.

Each program unit generally contains two parts: a declarative part, which defines the logical entities to be used in the program unit, and a list of statements, which define the execution of the program unit.

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

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

If and case statements allow the selection of an enclosed statement list based on the value of a condition or expression at the head of the statement. An assertion statement states that some correctness condition must hold whenever control reaches that point in a program. An exception statement explicitly raises a special situation requiring the action of an exception subprogram.

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

Certain statements are only applicable to paths. An inner statement specifies that a set of paths may begin execution. A connect statement specifies that a path is ready to connect with another path through one of its boxes. A local path request specifies that the path is ready for connection from another path.

Whenever a rendezvous is achieved between a local path request and a connect statement, any specified data transfer takes place, and both the local path and the connecting path may continue.

Every element in the language has a type, which defines its logical properties and the operations that can be performed on elements of the type. There are two basic classes of types: scalar types and composite types.

The scalar types INTEGER, BOOLEAN, and CHARACTER are predefined. Scaled types provide a means of performing exact numerical computation, without the restriction to integer values. Real types provide a means of performing floating point computations, which are necessarily approximate. Enumeration types provide a means for users to define problem dependent types with discrete values.

Composite types allow definitions of structured collections of related elements. The composite types in the language are array structures, record structures, and accesses to record structures that are allocated dynamically. A family of record structures may be defined by associating a record type with a variant part.

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

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

Finally the language includes facilities for conditional compilation and for generic program units.

1.3 Sources

In his initial preface to the Pascal report, Niklaus Wirth stated "the choice of what is to be omitted from a new language is in practice much more critical than the choice of what is to be included. The decision to omit a feature requires not only familiarity with this feature (and knowledge how to live without it) but the courage to face the inevitable criticism of its absence in the new language in spite of its presence in another existing language."

This proolem existed in this design, although to a much lesser degree than usual because of the Ironman requirements. These requirements often simplified the design process by permitting us to concentrate on the design of a logical system satisfying a well defined goal, rather than on the definition of the goals themselves.

Another significant simplification of our design work resulted from earlier experience acquired by several successful Pascal derivatives developed with similar goals. These are the languages Euclid, Lis, Mesa, Modula, Sue, and CS4. Many of the key ideas and syntactic forms developed in these languages have a counterpart in the Green language. We may say that whereas these previous designs could be considered as genuine research efforts, the Green language is the result of a project in language design engineering, in an attempt to develop a product that represents the current state of the art.

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

2. LEXICAL ELEMENTS

This section defines the lexical elements of the language.

2.1 Characters

All lexical elements may be composed from the 64 character subset of ASCII. These characters are grouped as follows:

(a) Alphabetic characters
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) Numeric characters
0 1 2 3 4 5 6 7 8 9

(c) Special characters
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^

(d) The underscore character

_

and the space character.

2.2 Identifiers

An identifier is formed by a sequence of alphabetic and numeric characters, the first being alphabetic. An underscore may be inserted between parts of an identifier. An identifier must fit on a single line, and all characters are significant. An identifier that has been declared is generally referred to as a name, with a prefix designating its use, e.g. variable_name or type_name.

Examples:

```
COUNT        X      LINE_COUNT  GET_SYMBOL
SNOBOL_4     X1     PAGE_COUNT  STORE_NEXT_ITEM
```

2.3 Numbers

There are three classes of numbers: integers, scaled numbers, and real numbers. An underscore may be inserted between parts of a number, but is not considered significant.

Integers are formed by a sequence of numeric characters.

Examples:

```
12      0      1977      123456      123_456
```

A scaled number is written as an integer or as a sequence of numeric characters with a medial decimal point.

Examples:

```
12.0    0.0    123.456    10_000.1    1200
```

A real number is formed by appending the letter E and an exponent to an integer or a scaled number. An exponent is an integer optionally prefixed by a + or – sign.

Examples:

```
12.0E10    0E0    1E-6    3.14159_26535E0
```

Non decimal integers with base 2,4,8, or 16 are written as a sequence of numeric characters followed by a # and a base. For hexadecimal numbers (base 16), the alphabetic characters A through F may be used with their conventional meaning, provided that the number begins with a numeric character.

Examples:

```
0110011#2    1777#8    2FEEE#16    0FFF#16
```

2.4 Character Strings

A character string is formed by a sequence of characters enclosed by quote marks. Strings of length one also denote literals of character types. If a string contains a quote mark, the quote mark must be written twice. Each string must fit on a single line. Multiple line strings may be formed using the concatenation operator &.

Examples:

```
"A"    "*"    "A SMALL STRING"
"FIRST PART OF A STRING THAT"  &
" CONTINUES ON ANOTHER LINE."
```

2.5 Comments and Pragmats

Comments may be placed within programs. A stand alone comment starts with the characters -- and is terminated by the end of the line. This form of comment may not appear within an expression or statement. An embedded comment is enclosed by left and right square brackets. Embedded comments may not cross line boundaries. Comments are totally ignored by the translator; their sole purpose is the enlightenment of the human interpreter of the program. For readability of this manual, comments will be written with both upper and lower case letters.

Pragmats (for pragmatics) are used to convey information to the translator. They start with the keyword **pragmat** and are terminated by the end of the line. A pragmat may not appear within a simple statement or within a declaration.

Examples of comments:

```
end  [GET_SYMBOL],
--   a stand alone comment
--   and its continuation.
```

Examples of pragmats:

```
pragmat  NO_LIST                  suppress listing
pragmat  LIST                     restore listing
pragmat  OPTIMIZE TIME            optimization specification
pragmat  INCLUDE COMMON_TEXT      include text file
pragmat  DEBUG                    set debugging mode
```

2.6 Attribute Qualifiers

Attribute qualifiers denote attributes of program constructs. An attribute qualifier is formed by prefixing one or more occurrences of the character ' to an identifier. Their use is described with the corresponding language constructs. Since attribute qualifiers always contain a ' character, their identifiers need not be reserved.

Examples:

```
DATE'SIZE        REAL'PRECISION        SYSTEM'CLOCK     A''LAST
```

2.7 Reserved Keywords

Language constructs may contain reserved keywords. These words may not be used as identifiers, and are listed below. For readability of this manual, the keywords appear with lower case letters in boldface; in actual programs they may be entered with upper case ASCII letters.

abs	**declare**	**if**	**or**	**scale**
access	**definition**	**import**	**others**	**separate**
algorithm	**delay**	**in**	**out**	**select**
alignment	**div**	**inline**		**send**
all		**inner**	**packing**	**subtype**
and		**interrupt**	**parameter**	
array	**else**	**is**	**path**	
assert	**elsif**		**pragmat**	**then**
at	**end**	**loop**	**precision**	**type**
	exception		**private**	
	exit	**mod**	**procedure**	
begin				**until**
bits		**new**	**raise**	**use**
box	**for**	**none**	**range**	
	function	**not**	**receive**	
		null	**record**	**when**
case			**repeat**	**while**
connect	**generic**	**of**	**return**	
constant	**goto**	**only**	**reverse**	**xor**

2.8 Spacing Conventions

Spaces may be inserted freely between lexical elements. Except for comments and pragmats, an end of line is equivalent to a space. At least one space must appear between two identifiers (reserved or not) that are not separated by a special character.

2.9 Syntax Notation

In the remaining chapters, a simple variant of Backus-Naur form is used to describe the context free syntax of the language. In particular,

(a) Lower case words, possibly containing medial underscore, denote syntactic categories, e.g. adding_operator

(b) Boldface words denote keywords in the language, e.g. **array**

(c) Square brackets enclose optional items, e.g. **return** [expression]

(d) Braces enclose items repeated zero or more times. For example, a list of identifiers is defined as:

identifier_list : : = identifier { ,identifier }

3. DECLARATIONS AND TYPES

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

3.1 Declarations

A declaration associates a name with a language construct. There are several kinds of declarations.

```
declaration : : =
        element_declaration        |  type_declaration
    |   subtype_declaration        |  access_type_declaration
    |   subprogram_declaration     |  path_declaration
    |   definition_declaration     |  generic_instantiation
    |   variant_part               |  null;
```

A null declaration introduces no new names; it may be used, for example, to define a record variant with no components. Declarations for elements, types, subtypes, and access types are described here. The remaining declarations are described in later sections.

3.2 Element Declarations

Element declarations introduce constants and variables.

```
element_declaration : : =
        variable_declaration       |   renaming_declaration
    |   constant_declaration       |   deferred_constant_declaration

variable_declaration : : =
    identifier_list: type [: = expression] ;

renaming_declaration : : =
    identifier: type == variable;

constant_declaration : : =
    identifier: constant [type] ·  · ression;

deferred_constant_declaration : : =
    identifier: constant type;
```

A variable declaration associates one or more identifiers denoting new variable names with a type. The declaration may specify an initial value for the variables. Each initialization is equivalent to an assignme it statement performed immediately after the declaration.

A renaming declaration associates a local name with a variable. The local name can be used as a shorthand for the variable, e.g. in references to an array or record structure.

A constant declaration associates a name with a value specified by an expression. The value is computed when the constant declaration is elaborated. The type of the constant may be omitted when the value is a literal whose type is known.

A deferred constant declaration specifies the name and type of a constant whose value is computed in specially restricted contexts.

Examples:

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

ANCESTOR           : PERSON == JOHN.FATHER.MOTHER;

ACCURACY           : constant = 1E-30;
LIMIT              : constant INTEGER = 10_000;

NULL_ENTRY         : constant ENTRY;
```

3.3 Type and Subtype Declarations

A type specifies a set of properties for elements of the type. A type declaration associates a name with a type.

```
type_declaration    : : = type identifier = type_definition;

type_definition     : : = type  |  private [parameter]

type : : =
        simple_type_definition [constraint]
    |   array_type     |     record_type

simple_type_definition  : : =  scalar_type  |  type_denotation

type_denotation  : : =  type_name  |  subtype_name  |  attribute

constraint  : : =
    scalar_constraint  |  array_constraint  |  record_constraint

subtype_declaration  : : =
    subtype identifier = type_denotation [constraint] ;
```

A subtype declaration associates a name with a parent type whose properties may be limited by some constraint. The use of **private** as a type definition is explained in the section on definition modules.

3.4 Scalar Types

Scalar types describe discrete values and the real numbers. Discrete types may be used for indexing. The scalar type names INTEGER, BOOLEAN, and CHARACTER are predefined discrete types. Other types may be declared by the user.

```
scalar_type  : : =  discrete_type  |  real_type  |  (range)

discrete_type  : : =  scaled_type  |  enumeration_type

scalar_constraint  : : =  range (range)

range  : : =  simple_expression . . simple_expression
```

A scalar constraint is specified by giving a range that describes a subset of values of the parent type. The range L . . R describes the subset of values from L to R inclusive. A scalar type given as a range is equivalent to giving the parent type of the expressions defining the range, with the range as a constraint.

The functions SUCC and PRED are predefined on all discrete types for which there is an implied ordering. They return the next higher or lower value in the range of values for the type. In addition, for an ordered discrete subtype or type T, the attributes T'FIRST and T'LAST denote the minimum and maximum values of the type.

3.4.1 Integer types

The predefined type name INTEGER denotes a subset of the whole numbers. The range of integer numbers is implicitly limited by the representation adopted by an individual implementation. Derived types may be obtained by imposing a range constraint.

Examples:

```
type PAGE_NUM     =  INTEGER;
type LINE_SIZE    =  (1 . . MAX_LINE_SIZE);

subtype SMALL_INT    =  INTEGER    range (-10 . . 10);
subtype COLUMN_PTR   =  LINE_SIZE  range (1 . . 10);
```

3.4.2 Scaled types

Scaled types provide a means of performing exact numeric calculations on non-integer values. Corresponding to every scaled type, there is a constant scale factor. All quantities of the type are an integer multiple of the scale factor. The scale factor is specified in the type declaration and has a value which is either an integer or the reciprocal of an integer.

```
scaled_type  : : =  scale simple_expression
```

The value of the expression defining the scale must be known at translation time. Within the scope of the type, the scale factor of a scaled type T can be accessed with the attribute T'SCALE.

Examples:

```
type TICK   =  scale  1 / / 60     range (0 . . 3600);
type VOLT   =  scale  1 / / 1 000  range (0 . . 1.5);
type JOULE  =  scale  1000         range (0 . . 1_000_000);
```

3.4.3 Real types

Real types provide a means of performing floating point computations, which are necessarily approximate. The relative precision of a real number is specified in the type declaration, and is used to bound the errors inherent in floating point computation.

```
real_type  : : =  precision simple_expression
```

The value of the expression defining the precision must be known at translation time. Within the scope of a real type T, the precision of the type can be accessed with the attribute T'PRECISION.

Examples:

```
type LONG_REAL      =  precision  1E-40;
type COEFFICIENT    =  precision  1E-10 range (-1E0 . . 1E0);
```

3.4.4 Enumeration types

An enumeration type defines a set of values by listing the values of the type. These values are unordered if the separator ! is used; they are listed in increasing order if the separator , is used.

```
enumeration_type  : : =
        (enumeration_value  { ! enumeration_value })
    |   (enumeration_value  { < enumeration_value })

enumeration_value  : : =  identifier  |  character
```

Examples:

```
type SUIT          =  (CLUBS < DIAMONDS < HEARTS < SPADES);
type HEX_LETTER    =  ("A" | "B" | "C" | "D" | "E" | "F");
type DAY           =  (MON | TUE | WED | THU | FRI | SAT | SUN);

subtype WEEK_DAY   =  DAY range (MON .. FRI);
subtype REST_DAY   =  DAY range (SAT .. SUN);
```

3.4.5 Boolean and character types

The enumeration type name BOOLEAN is predefined. It contains the two unordered values TRUE and FALSE.

The enumeration type name CHARACTER is a standard library defined type. The allowed characters and their ordering are defined by a given implementation.

3.5 Array Types

An array is a collection of elements of the same element type. The elements of an array are designated by indices.

```
array_type : : = array (index {, index}) of type

index : : = range_denotation    |    *

range_denotation : : = range    |    type_denotation

array_constraint : : = (range. denotation {,range_denotation})
```

The type of an array is given by the number of its indices and the type of its elements. An index has a specified range, which is not part of the type of the array. The index * denotes an arbitrary range of any discrete type.

When an array type name has been defined in a type declaration, an array constraint may be associated with the name in order to specify the actual ranges of the indices.

For an array type T, the attributes T'RANGE, T'FIRST, and T'LAST denote the range of the first index, its lower bound, and its upper bound. Similarly, the attributes T' 'RANGE, T' 'FIRST, and T' 'LAST serve the same role for the second index, and so forth.

Examples:

```
type T = array (*,*) of BOOLEAN;

A : T(1 .. 10, 1 .. 100);
B : array (1 .. 10, 1 .. 100) of BOOLEAN;

A'FIRST      [value is 1]
A''LAST      [value is 100]
```

3.5.1 Dynamic arrays

The range of each index for an array must be known at the time of array allocation. If the range of an index is not computable at translation time, the array is considered as a dynamic array. Dynamic arrays may also appear in records denoted by access types.

3.5.2 Array aggregates and strings

An array aggregate denotes a value for an array, i.e. is a constructor for an array. Indices are denoted by selections and element values by expressions.

```
array_aggregate : : =  character_string
    |  [type_name] (element_specification {, element_specification})

element_specification : : = selection: expression

selection : : = selected_value {| selected_value}

selected_value : : =
    number  |  enumeration_value  |  range_denotation  |  others
```

A selection specifies a set of individual values of a discrete type. A range denotation given in a selection stands for all values in the range. The keyword **others** denotes all other elements not specified in previous selections. Selections are also used in case statements and record variants.

A character string is considered as an array aggregate. A string of N characters for $N \geq 1$ is an array of a character type. Its range is 1 . . N.

Multi–dimensional array aggregates are treated as arrays of arrays.

Examples:

```
type TABLE = array (1 . . 10) of INTEGER;
type LINE  = array (1 . . MAX_LINE_SIZE) of CHARACTER;

A : TABLE : = (1 ! 2: 1, others: 0);

BLANK_LINE: constant LINE = (1 . . MAX_LINE_SIZE: ' ');
```

3.5.3 Sets

The predefined type name SET denotes one-dimensional boolean arrays.

```
type SET = array (*) of BOOLEAN;
```

Boolean valued operators are applicable to boolean vectors, i.e. one dimensional boolean arrays. These operators perform the corresponding operations on an element by element basis. Array aggregates may be used to denote set values.

Examples:

```
type WEEK = SET (DAY);                ! set type]
(TUE!THU: TRUE, others: FALSE)        sc  2 days]

(X and Y) = X       [test if X is a subset of Y]
X (E) = TRUE        [test if E is an element of X]
```

3.6 Record Types

A record type defines a structure with several components. The names and types of the components are introduced in the element declarations of the component list. A record type may include a variant part and hence define a family of structures.

```
record_type : : = record component_list end record

component_list : : = {element_declaration } [variant_part]

variant_part : : = case discriminant of {variant } end case;

discriminant : : = variable_name

variant : : = when selection = > component_list
```

An element declaration defining a record component may specify an initial value for the component.

Example:

```
type DATE =
    record
        DAY   : (1..31);
        MONTH : MONTH_NAME;
        YEAR  : (0..2000);
    end record;
```

3.6.1 Constant components, unassignable components, and variant parts

A record component declared as a constant serves to denote a constant valued component that has the same value for all records of the type.

A record component declared as a deferred constant is an unassignable component. Its value may only be set by a complete record assignment.

A record type with a variant part specifies a family of record structures. A variant part is discriminated by a previously declared component called the discriminant (or tag field). Each variant defines the components for the corresponding value of the discriminant. The discriminant must be declared as a deferred constant and hence is unassignable.

Example:

```
type PERIPHERAL =
    record
        STATUS: (OPEN | CLOSED);
        UNIT  : constant (PRINTER | DISK | DRUM);
        case UNIT of
            when PRINTER =>  LINE_COUNT: (1..PAGE_SIZE);
            when others  =>
                CYLINDER :CYLINDER_INDEX;
                TRACK    : TRACK_NUMBER;
        end case;
    end record;
```

3.6.2 Record aggregates and record constraints

A record aggregate denotes a value for a record, i.e. is a constructor for a record. The value is constructed by giving the values of its components.

```
record_aggregate : : = [type_name] (component_specification
    {,component_specification})

component_specification : : =
    component_name {|component_name}: expression

record_constraint : : = record_aggregate
```

If a record type contains a variant part, the selected component names must correspond to the specified value of the discriminant.

If a previously declared record type contains several variants, a record constraint may be used to constrain record variables or subtypes to a specified variant. The record constraint specifies the value of the selected variant. It is expressed in the form of a record aggregate where values are provided only for discriminants.

Examples of record aggregates:

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

Example of record constraint:

```
subtype  DISK_DEVICE = PERIPHERAL(UNIT: DISK);
```

3.7 Access Types

Normal record variables declared in a program are accessible by their identifier. They exist during the lifetime of the declarative part to which they are local and are hence said to be static. In contrast, a variable of an access type is used to designate a record that is allocated dynamically.

```
access_type_declaration : : = access type identifier == type;
```

Access to a dynamic record is achieved via an access variable which may be set by an allocation statement or by assignment of another access variable. The value of an access variable that does not designate a dynamic record is denoted by **none**.

Each access type declaration implicitly defines a collection of dynamically allocated records that can be referenced by variables of the access type. Given record may be designated by more than one variable of the access type. Components of the records of an access type may belong to the same access type.

A representation specification (see section 10) may be used to specify the storage space to be (statically) reserved for the collection of records associated with an access type.

Examples:

```
access type PERSON ==
    record
        NAME    : STRING;
        AGE     : INTEGER;
        MOTHER  : PERSON;
        FATHER  : PERSON;
    end record;

access type LIST_ITEM ==
    record
        VALUE : INTEGER;
        SUCC  : LIST_ITEM;
        PRED  : LIST_ITEM;
    end record;
```

3.8 Type Conformity

Each type, subtype, variable, and constant has a base type, which is the fundamental property used to check type conformity.

Declarations of distinct type names always denote distinct base types, even if their definitions are identical. Type constraints do not alter the base type. The base type of a subtype is that of its parent type. The base type of a variable or constant is that of the type appearing in the declaration.

Declarations involving unnamed types obey the following rules:

(a) If the type is given as an enumeration or as a record, the base type is distinct from any other enumeration or record type, even if their definitions are textually identical.

(b) If the type is given as a range, the base type is that of the expressions defining the range.

(c) Two real or scaled types have the same base types if their precisions or scale factors are the same.

(d) Two array types have the same base type if they have the same number of dimensions and if their elements have the same base type and constraints.

If a type A is defined in terms of another type name B

```
type A = B;
```

then A and B are two different types that share the same logical properties but not necessarily the same representation. Explicit conversions between related types like A and B are possible and must be written as typed expressions.

3.9 Declarative Parts

Each program unit may contain a declarative part specifying its declarations and other local information.

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

body : : = subprogram_body | definition_module_body | path_body
```

An identifier declared within a program unit has a scope, which consists of the unit in which the identifier is declared and all inner units that do not redeclare the same identifier. An identifier is said to be "local" to the unit in which it is declared, and "global" to all inner units that do not redeclare the same identifier.

An import clause is used to import identifiers of definition modules. Representation specifications define particular type representations. The bodies of subprograms, definition modules, and paths declared in the declaration list appear at the end of the declarative part. These constructs are defined in later chapters.

4. VARIABLES AND EXPRESSIONS

4.1 Variables

A variable denotes a stored value of a given type. It may be a name denoting a scalar value, an array, or a record. Alternatively, it may denote an element of an array, a slice of an array, or a record component.

```
variable ::=
    variable-name | array_element | slice | record_component

array_element ::= variable(expression {,expression})

slice ::= variable(range_denotation)

record_component ::= variable.component_name | variable.all
```

For array elements, the expressions denote index values. For array slices, the specified range denotes a contiguous sequence of index values.

Record components may denote either components of static record variables or components of dynamic records designated by access variables. The qualifier **all** denotes all components of a dynamic record. A record component within a record variant can only appear in contexts where the particular variant is known.

Examples:

```
PRESSURE                    APPOINTMENT.DAY
MATRIX(I,J+1)               STACK(TOP).NAME
TABLE(1 .. N)               NEXT.SUCC.VALUE
```

4.2 Scalar Values and Attributes

A scalar value denotes a value of a scalar type. In addition, scalar values are used to denote atttributes of declared entities.

```
scalar_value ::= number | enumeration_value | attribute

attribute ::= denotation'attribute_qualifier

denotation ::=
    name | variable | path_denotation | box_denotation
```

An attribute qualifier specifies a property of some denoted program construct. An attribute qualifier for a type is also an attribute qualifier for all variables of the type. Specific attribute qualifiers are described with the corresponding language constructs.

Examples of attributes:

```
X'PRECISION      [the relative precision of a variable]
INDEX'FIRST      [the lower bound of a range]
DATE'SIZE        [the number of bits in a record]
```

4.3 Expressions

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

```
expression ::=
        simple_expression [relational_operator simple_expression]
    |   simple_expression is [not] range_denotation

simple_expression ::= [simple_expression adding_operator] term

term ::= [term multiplying_operator] factor

factor ::= [unary_operator] primary

primary ::= variable | scalar_value | array_aggregate
    | record_aggregate | function_call | (expression)
    | qualified_expression | none

function_call ::= subprogram_call
```

The type of an expression depends on the type of its components, as described below.

Examples of primaries:

```
VOLUME                  [variable]
4.0                     [number]
(1..10: 0)              [array aggregate]
SINE(X)                 [function call]
(LINE_COUNT + 10)       [parenthesized expression]
REAL(I,J)               [qualified expression]
```

Examples of expressions:

```
VOLUME                              [primary]
-4.0                                [factor]
not DESTROYED                       [factor]
LINE_COUNT mod PAGE_SIZE            [term]
B*B - 4EO*A*C                       [simple expression]
(INDEX = 0) or ITEM_HIT             [simple expression]
PASS_WORD(1 .. 5) = "JAMES"         [expression]
X is 1 .. 10                        [expression]
```

4.4 Operators

The operators in the language are grouped into four classes

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

adding_operator       ::=  +  |  -  |  or  |  xor  |  &

multiplying_operator  ::=  *  |  /  |  //  |  mod  |  div  |  and

unary_operator        ::=  +  |  -  |  not  |  abs
```

These operators have a precedence that specifies the order of evaluation within an expression. Unary operators are applied first, multiplying operators second, adding operators third, and relational operators last. Sequences of operators of the same precedence are evaluated from left to right.

The use and meaning of the operators are given below. All binary operators apart from is and * must be applied to operands of the same type. In particular, to perform arithmetic on two numeric values of differing types, one of the values must be explicitly converted to the type of the other.

4.4.1 Relational operators and is

The relational operators and is all return boolean values.

Operator	Operation	Operand Types	Result Type
is [not]	range membership	any scalar type and corresponding range	boolean
= /	equality and inequality	any type	boolean
< <= > >=	test for relative ordering	any ordered type	boolean

Note that equality and inequality are defined for any two objects of the same type.

4.4.2 Adding operators

All adding operators return a result of the same type as the operands.

Operator	Operation	Operand Types	Result Type
+ –	addition and subtraction	numeric	same numeric type
or xor	inclusive and exclusive disjunction	boolean, boolean vector	boolean, boolean vector
&	concatenation	one-dimensional arrays, slices, array elements, and characters	one-dimensional array of element type

The operator & concatenates the elements in one array to those in another array. For strings, this operation results in conventional string concatenation.

4.4.3 Multiplying operators

Operator	Operation	Operand Types	Result Type
*	multiplication	numeric	numeric
/	real division	real	real
//	scaled division	integer	scaled
div	integer division	integer	integer
mod	modulus	integer	integer
and	conjunction	boolean, boolean vector	boolean, boolean vector

The operator * denotes mathematical multiplication. It takes two integer or real operands of identical type and gives a result of the same type. In addition, a scaled operand can be multiplied by an integer to give a result whose type is the same as the scaled operand.

The operator / denotes mathematical division and is defined only for real types. The resulting type is the same as that of its operands.

The operator / / denotes mathematical division and is defined only for operands of integer type. The result is of scaled type, where the particular scale factor depends upon the operands.

The operators div and mod denote integer division with truncation and the remainder after integer division. These operators are defined only for integer operands.

Examples:

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

M  :  scale 1 / / 3 := 4 / / 3;
N  :  scale 1 / / 3 := 5 / / 3;

X  :  precision 1E-6 := 1E0;
Y  :  precision 1E-6 := 2E0;
```

Expression	Value	Type
I * J	2	same as I and J
J * M	8//3	same as M
X / Y	0.5E0	same as X and Y
K div J	1	same as K and J
K mod J	1	same as K and J
M + 1//3	5//3	same as M
N + K//3	8//3	same as N

4.4.4 Unary operators

Unary operators are applied to a single operand.

Operator	Operation	Operand Type	Result Type
+ –	identity and negation	numeric	same numeric type
not	negation	boolean, boolean vector	boolean, boolean vector
abs	absolute value	numeric	same numeric type

4.5 Qualified Expressions

A qualified expression is used to convert an expression to another type, to state the type of an expression explictly, or to constrain an expression to a given subtype.

```
qualified_expression  : : =
    typed_expression    |   constrained_expression

typed_expression  : : =  type_name(expression)

constrained_expression  : : =  subtype_name(expression)
```

4.5.1 Type conversions

For numeric expressions, a typed 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. The nearest value of the required type is the value after conversion. If two values are equidistant from the expression value, then the larger value is chosen.

Typed expressions can also be used for type conversions between related types with identical logical properties. No other type conversions are permitted.

Examples of numeric type conversion:

```
REAL(2*1)           {value is converted to real}
INTEGER(1.6)        {value is 2}
INTEGER(-0.5)       {value is 0}
```

Example of conversion between related types:

```
type A_FORM = B_FORM;
X :  A_FORM;
Y :  B_FORM;

X : = A_FORM(Y);
```

4.5.2 Type specification of values

The same element may appear in two enumeration types. In these cases, and whenever the type of a literal or aggregate is not known from the context (e.g. an actual parameter of an overloaded procedure), a typed expression may be used to state the type explictly.

Examples:

```
type MASKING_CODE = (FIX  | DEC  | EXP  | SIGNIF);
type INSTR_CODE   = (CLA  | DEC  | TNZ  | SUB);

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

4.5.3 Constrained expressions

An expression of a given type may have values that are not necessarily in one of its subtypes. A qualified expression with a subtype name specifies that the value of an expression must belong to the subtype. If it does not, an exception condition is raised.

Examples:

```
subtype SMALL_INT = INTEGER range (-10 .  10);
VALUE : INTEGER;
INDEX : SMALL_INT;

READ (VALUE);
INDEX := SMALL_INT(VALUE);
```

5. STATEMENTS

Statements cause actions to be performed. Statements in a list of statements are executed in sequence until a transfer statement is encountered.

A statement may be simple or compound. A simple statement contains no part that constitutes another statement

```
statement_list : : = { [ label ] statement }

statement : : = simple_statement | compound_statement
    | transfer_statement

simple_statement : : = assignment_statement | allocation_statement
    | subprogram_call_statement | assert_statement
    | synchronization_statement | inline_statement
    | null ;

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

transfer_statement : : = loop_exit_statement
    | return_statement | exception_statement
    | goto_statement

label : : = <<identifier>>
```

Execution of a null statement results in no action. Synchronization and select statements are described in the section on parallel processing. Inline statements are described in the section on subprograms. Exception statements are described in the section on exception handling. The remaining statements are described here.

5.1 Assignment Statements

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

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

The variable and the expression must be of the same parent type and the value of the expression must satisfy any constraints applicable to the variable. If the constraints cannot be checked during translation, an execution-time check shall be provided by the translator. This check will result in an exception condition if the expression value does not satisfy the constraint. If the exception is suppressed the translator will omit the checks.

Examples:

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

Examples of constraints:

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

I    : =    J;    -- identical ranges
K    : =    J;    -- compatible ranges
J    : =    K;    -- can only be checked during execution
```

5.1.1 Array and slice assignments

For an assignment to an array or an array slice variable, the expression must denote a value with the same number of elements. For slice assignments where the array name of the slice variable also appears in the expression, overlapping of index ranges is forbidden.

Examples:

```
A    :   array (0 .. 30) of CHARACTER;
B    :   array (1 .. 31) of CHARACTER;

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

5.1.2 Record assignments

If a record variable has been declared with a record constraint, the variant assigned must have a discriminant value prescribed by the constraint.

Examples:

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

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

5.2 Allocation Statements

An allocation statement specifies the dynamic creation of a record to be designated by an access variable.

```
allocation_statement    : : =
        variable    : =   new record_aggregate;
    |   variable    : =   new typed_expression;
```

Storage for a record is allocated with the collection associated with the access type. The name of this access type must appear explicitly after the keyword **new** in either case. The value of the record aggregate or typed expression is assigned to the new record, and the access variable is made to designate the new record.

Examples:

```
ELEMENT    : =   new LIST_ITEM (VALUE: 0, SUCC: none, PRED: none);
DOUBLE     : =   new PERSON (ME, all);
```

5.3 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 declaration. An actual parameter is either a variable or an expression.

```
subprogram_call_statement    : : =   subprogram_call;

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

parameter_association    : : =
    input_assocation   |   output_association   |   access_association

input_association    : : =   [formal_parameter    : = ]    expression

output_association   : : =   [formal_parameter    = : ]    variable

access_association   : : =   [formal_parameter    == ]    variable

formal_parameter     : : =   identifier
```

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 parameter is explicitly given in the call. Named parameters may be given in any order.

Positional parameters and named parameters may be used concurrently with positional parameters occuring first at their normal position, i.e. once a named parameter is used the rest of the call must use only named parameters.

Examples:

```
RIGHT_SHIFT;

SEARCH_STRING (STRING, CURRENT_POSITION, NEW_POSITION);

PLOT (CURVE       := SINE,
      LOWER_BOUND := N1,
      UPPER_BOUND := N2);

RE_ORDER_KEYS (NUM_OF_ITEMS, KEY_ARRAY == RESULT_TABLE);
```

5.3.1 Actual parameter associations

There are three forms for specifying actual parameters.

(a) Input parameter association.
The corresponding formal parameter must have the mode **in**, and acts as a local constant whose value is provided by the actual parameter prior to execution of the subprogram body.

(b) Output parameter association.
The corresponding formal parameter must have the mode **out**, and acts as a local variable whose value is assigned to the actual parameter upon return from the subprogram body.

(c) Access parameter association.
The corresponding formal parameter must have the mode **access**. Within the subprogram body, the formal parameter enables read and write access to the corresponding actual parameter.

Constantness for the in mode must be interpreted transitively. For example, the elements of an input array parameter may not be updated, and an input parameter may not be updated by calls to other subprograms.

5.3.2 Omission of actual parameters

An input parameter may be omitted from the list of 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. Similarly, an output parameter may be omitted if the value returned is not used in the calling program.

Example:

```
ACTIVATE: procedure (TASK    :  in TASK_NAME;
                     AFTER   :  in TASK_NAME   := NO_TASK;
                     DELAY   :  in REAL        := 0E0;
                     PRIOR   :  in BOOLEAN     := FALSE);

ACTIVATE (X);
ACTIVATE (X,   AFTER := Y);
ACTIVATE (X,   DELAY := 5E0*MINUTE,   PRIOR := TRUE);
```

5.3.3 Restrictions on subprogram calls

The type of each actual parameter must agree with that of the corresponding formal parameter. To prevent aliasing (i.e. multiple access paths to the same variable), the same variable name cannot be used for more than one actual output or access parameter.

5.4 Return Statements

A return statement terminates execution of a subprogram or a path. For functions, a return statement must include an expression whose value is the result of the function.

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

Examples:

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

5.5 If Statements

An if statement allows the selection of a statement list based on the truth value of one or more conditions.

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

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

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

Examples:

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

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

5.5.1 Short circuit conditions

A condition may appear as a sequence of boolean expressions separated by **and then**. In such case, evaluation of the expressions proceeds from left to right until one evaluates to false. The final value, true or false, is the value of the condition. Similarly, for expressions separated by **or else**, evaluation stops as soon as an expression evaluates to true.

Examples:

```
if (NEXT /= none) and then (NEXT.AGE < 18) then
  MINOR := TRUE;
end if;

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

5.6 Case Statements

A case statement allows the selection of a statement list based on the value of an expression at the head of the case statement

```
case_statement    ::=
    case expression of {alternative} end case;

alternative    ::=  when selection => statement_list
```

Execution of a case statement results in execution of the statement list whose selection contains the value of the expression. A given selection value may appear in only one alternative. Selection values must be provided for all values of the type of the expression. Note that it is always possible to constrain an expression to a given subtype or to use the selection **others** to cover any remaining values.

Examples:

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

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

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

5.7 Assertion Statements

An assert statement introduces an assertion that must hold whenever control reaches that point in the program.

```
assert_statement   : : =   assert [condition] ;
```

Examples:

```
assert (Y2 - Y1) < EPSILON;
assert (INPUT_CHARACTER is "A" . . "Z");
assert [There exists an I such that A(I) > 0];
```

The assertion may be formulated as a condition or as a comment. The condition is treated as a comment whenever checking of assertions is suppressed. Otherwise, the condition is evaluated and an exception is raised if the condition does not hold (see section 9).

5.8 Loop Statements

A loop statement specifies that a statement list in a basic loop is to be executed repeatedly. Execution is terminated when either the iteration specification of the loop is completed or when a loop exit statement within the basic loop is executed.

```
loop_statement    : : =   [iteration_specification] basic_loop;

basic_loop    : : =   loop statement_list repeat

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

loop_parameter    : : =   identifier
```

In a loop statement with a for clause, the loop parameter is implicitly declared as a local each execution of the basic loop. The loop statement is terminated if the while expression is false or the until expression is true.

In a loop statement with a for clause, the loop parameter is implicitly declared as a local variable whose type is that of the elements in the range denotation. On successive loop iterations, the loop parameter is successively assigned values from the specified range. The values are assigned in increasing order unless the keyword **reverse** is present, in which case the values are assigned in decreasing order. Within the basic loop, the loop parameter acts as a constant whose value may not be changed.

Examples:

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

until BUFFER (I)  = ";" loop
    I    : =    I + 1;
repeat;

for I in BUFFER'RANGE loop
    BUFFER (I)    : =    BLANK;
    SQUARE (I)    : =    I*I;
repeat;
```

5.9 Loop Exit Statements

A loop exit statement causes explicit termination of a loop. It may contain a condition, in which case termination occurs only if its value is true.

```
loop_exit_statement    : : =    exit [when condition];
```

A loop exit statement may only appear in a basic loop.

Examples:

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

5.10 Blocks

A block introduces a new declarative part for a list of statements. Execution of a block results in elaboration of the declarative part followed by execution of the statement list.

```
block    : : =
    declare declarative_part begin statement_list end;
```

Identifiers declared in a declarative part follow the same scope rules as those for subprograms and definition modules, as described in later sections.

5.11 Goto Statements

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

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

The statement to which control is transfered must be labeled with the corresponding identifier. The designated statement must be within the same local scope as the goto statement. Transfer of control into a compound statement is not allowed.

Example:

```
<<COMPARE>>
    if A(I) < ELEMENT then
        if LEFT (I) > = 0 then
            I    : =    LEFT (I);
            goto COMPARE;
            . . .
        end if;
        . . .
    end if;
```

6. SUBPROGRAMS

This section and the following sections describe the rules for defining higher level program units. These include procedure and function subprograms, definition modules, parallel paths, and exception subprograms. Separate compilation of program units and generic program units are discussed in the section on overall program structure.

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

6.1 Subprogram Declarations

A subprogram declaration specifies the name of a subprogram, its nature, its formal parameters, the type of any returned value, and, possibly, a translation mode indicating whether it is separately compiled or generic.

```
subprogram_declaration ::=
    namer: [translation_mode] subprogram_nature formal_part;

subprogram_nature ::= procedure | function | exception

formal_part ::=
    [(parameter_definition {; parameter_definition})] [return type]

parameter_definition ::=
    identifier_list: mode type [ := expression]

mode ::= [in] | out | access

namer ::= identifier | character_string

translation_mode ::= separate | generic_clause
```

Examples:

```
TRAVERSE_TREE  : procedure;
RIGHT_INDENT   : procedure (MARGIN: out LINE_POSITION);
COMMON_PRIME   : function (N,M: INTEGER) return INTEGER;
```

6.2 Formal Parameters

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

in The parameter acts as a local constant whose value is set equal to that of the corresponding actual parameter upon call to the subprogram.

out The parameter acts as a local variable whose value is assigned to the corresponding actual parameter upon return from the procedure.

access The parameter acts as a variable and may be used for read and write access to the corresponding actual parameter.

If no mode is explicitly given, the mode **in** is assumed.

For **in** parameters, the parameter definition may also include a specification of an expression that is implicitly assigned to the parameter if no explicit value is given in the call. This expression may either be an expression computable at translation time or a variable.

Examples:

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

ACTIVATE:     procedure (TASK   : in TASK_NAME;
                         AFTER  : in TASK_NAME := NO_TASK;
                         DELAY  : in REAL := 0E0;
                         PRIOR  : in BOOLEAN := FALSE);
```

6.3 Subprogram Bodies

A subprogram body specifies the execution of a subprogram.

```
subprogram_body ::=
    [inline] subprogram_nature namer formal_part;
    declarative_part [begin statement_list] end;
```

Upon each call to a subprogram, the correspondence between actual and formal parameters is established (see section 5.3), the declarative part is elaborated, and the statements of the body are executed. Upon successful completion of the body, return is made to the caller.

Subprogram bodies marked as **inline** are expanded in line at each call. The text of each actual parameter replaces the corresponding formal parameter. Identifiers other than actual parameters are interpreted in the scope of the subprogram declaration. Thus the meaning of a subprogram is not changed by the prefix **inline**. Inline subprograms cannot be separately compiled.

If a subprogram body appears in the same declarative part as its declaration and is not referenced in previous subprograms, the subprogram declaration may be omitted. In such a case the body acts as a substitute for the subprogram declaration. A subprogram body without a statement list is used when the statement list is separately compiled (see section 11.3).

Example:

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

6.4 Function Subprograms

A function is a subprogram that computes a value. A function declaration may only have in parameters and contains 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.

Side effects to variables accessible at the function call are not allowed. In particular, variables that are global to the function body may not be updated in the function body.

If a function parameter belongs to an access type, the parameter must be viewed as providing access to the complete collection of dynamic records. As a consequence, within the function body there can be no alteration to any component of a record designated by an access variable.

Example:

```
function DOT_PRODUCT (X,Y: REAL_VECTOR) return REAL;
    SUM: REAL := 0E0;

begin
    assert (X'FIRST = Y'FIRST);
    assert (X'LAST  = Y'LAST);
    for I in X'RANGE loop
        SUM := SUM + X(I)*Y(I);
    repeat;
    return SUM;
end [DOT_PRODUCT];
```

6.5 Overloading

Functions and procedures of a given name may have multiple definitions, each having parameters of differing types. In these cases the redefinition of a named function or procedure does not hide any previous definition. On subprogram calls, the actual definition used is that whose formal parameter types match those of the actual parameters.

6.5.1 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 functions and procedures, except that the character string must be one of the operators in the language. Overloading applies to both unary and binary operators. Overloading does not change the precedence of an operator. Overloading of the concatenation operator & is not allowed.

Examples:

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

6.6 Code Insertions

A machine code insertion can be achieved by a call to an inline procedure whose body only contains inline statements.

```
inline_statement ::= inline record_aggregate;
```

Each machine instruction appears as an inline record aggregate of a record type defining the corresponding instruction. Such record definitions will generally be available in a library definition module for each machine. A procedure that contains an inline statement must contain only inline statements. Insertions of code written in other programming languages must be bracketed by special pragmats.

Examples:

```
M: MASK;
inline procedure SET_MASK;
    import INSTRUCTION_360;
begin
    inline SI_FORMAT (CODE: SSM, B: M'BASE, D: M'DISP);
end;

inline procedure ROOT;
begin
    pragmat FORTRAN BEGIN
    [FORTRAN text]
    pragmat FORTRAN END
end;
```

7. DEFINITION MODULES AND SCOPE RULES

Definition modules allow the specification of groups of logically related entities. In their simplest form they can represent pools of common data and type declarations.

In addition, definition modules can be used to describe packages of related subprograms and encapsulated data types, whose inner workings are concealed from their users.

Scope rules, the rules defining the entities that are visible at a given program point, are explained in this section for definition modules and other program units.

7.1 Specification of Definition Modules

The specification of a definition module generally includes its declaration and the specification of a definition module body. Its declaration may be omitted unless it contains a translation mode specifying the module as separately compiled or generic.

```
definition--declaration :: =
    identifier: [translation_mode] definition;

definition_module_body :: =
    definition identifier; declarative_part
    [private declarative_part]
    [algorithm declarative_part]
    [begin statement_list] end;
```

A variable declared in any of these declarative parts is said to be "own" to the definition module. Own variables remain allocated for the life time of the definition module. Elaboration of the declaration of the definition module results in the allocation of its own variables, the assignment of any initial values, and the execution of the statement list initializing the definition module.

Variables that are declared within local subprograms of the definition module are not own. Hence they do not retain their values from one call to the next.

Examples of definition declarations:

```
PLOTTING_DATA    : definition,
TABLE_MANAGER    : definition;
I_O_PACKAGE      : definition,
```

7.2 The Visible Part

The first declarative part of a definition module is called its visible part. The entities declared in the visible part are accessible to program units that import the definition name. A definition module containing only a visible part may be used to represent a group of common variables or a common pool of data and types.

Example of group of common variables:

```
definition PLOTTING_DATA;

    PEN_UP: BOOLEAN;
    CONVERSION_FACTOR: REAL;

    X_OFFSET, Y_OFFSET,
    X_MIN, X_MAX,
    Y_MIN, Y_MAX: scaled 1//100 range (0 .. 30);

    GRID_VALUE: array (1 .. 500) of
                    record
                      X_COORD: REAL;
                      Y_COORD: REAL;
                    end record;
end [PLOTTING_DATA];
```

Example of common pool of data items and types:

```
definition WORK DATA;
    type DAY_LENGTH = INTEGER range (0 .. 24);
    type DAY = (MON | TUE | WED | THU | FRI | SAT | SUN);

    WORK_HOURS    :   array (DAY) of DAY_LENGTH;
    NORMAL_HOURS  :   constant array (DAY) of DAY_LENGTH =
                          (MON .. THU: 8, FRI: 7, SAT | SUN: 0);
end [WORK_DATA];
```

7.3 Algorithm Part

The visible part of a definition may contain subprogram or path declarations. In such cases, their bodies may be grouped in an algorithm part, the declarative part after the keyword **algorithm**. The algorithm part may also include local declarations and local program units needed by these bodies.

In contrast to the entities declared in the visible part, the entities declared in the algorithm part are not accessible outside the definition module. As a consequence, a definition module with an algorithm part can be used for the construction of a package, where the logical operations accessible to the user are clearly isolated from the hidden internal entities.

Example of a package:

```
definition RATIONAL_NUMBERS;
    type RATIONAL =
        record
            NUMERATOR   : INTEGER;
            DENOMINATOR : (1 .. INTEGER'LAST);
        end record;
    EQUIV   : function (X, Y : RATIONAL) return BOOLEAN;
    ADD     : function (X, Y : RATIONAL) return RATIONAL;
    MULT    : function (X, Y : RATIONAL) return RATIONAL;

algorithm
    procedure SAME_DENOMINATOR(X,Y: access RATIONAL);
    begin
        [reduces X and Y to the same denominator]
    end;

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

    function ADD . . .
    function MULT . . .
end;
```

7.4 Private Part

The structural details of some declared types may be irrelevant to their logical use outside a definition module. This may be accomplished by stating in the visible part that a type is **private**. Types declared private in the visible part must be redeclared in full in the private part of the definition module.

For a private type, the only information available for external program units importing the definition module is the private type name. As a consequence, the only external operations permitted on variables of a private type are assignment and comparison of elements for equality or inequality.

If a type is declared as **private parameter**, not even assignment and comparison are permitted. Hence variables of such types may only be passed as parameters to subprograms declared in the visible part of the same definition module.

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

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

In general, private types may be used to define encapsulated data types. An example is given in Section 11.7.

Example:

```
definition I_O_PACKAGE;

    type FILE_NAME = private;
    NO_FILE: constant FILE_NAME;

    ASSIGN : procedure (F: out FILE_NAME);
    READ   : procedure (ELEM: out INTEGER; F: in FILE_NAME);
    WRITE  : procedure (ELEM: in  INTEGER; F: in FILE_NAME);

private
    type FILE_NAME = INTEGER range (0 .. 50);
    NO_FILE: constant FILE_NAME = 0;

algorithm
    type FILE_DESCRIPTOR = record ... end record;
    DIRECTORY: array (FILE_NAME) of FILE_DESCRIPTOR;
    ...
    procedure ASSIGN ...
    procedure WRITE ...
    procedure READ ...
end;
```

7.5 Example: A Table Management Package

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

```
definition TABLE_MANAGER;
    type ITEM =
        record
            ORDER_NUM  :  INTEGER;
            ITEM_CODE  :  INTEGER;
            ITEM_TYPE  :  CHARACTER;
            QUANTITY   :  INTEGER;
        end record;

    NULL_ITEM: constant ITEM =
        (ORDER_NUM: 0, ITEM_CODE: 0, ITEM_TYPE: " ", QUANTITY: 0);

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

algorithm
    SIZE: constant INTEGER = 2000;
    subtype INDEX = INTEGER range (0 .. SIZE);

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

    TABLE: array (INDEX) of INTERNAL_ITEM;
    FIRST_BUSY_ITEM:  INDEX := 0;
    FIRST_FREE_ITEM:  INDEX := 1;

    function FREE_LIST_EMPTY return BOOLEAN; ...
    function BUSY_LIST_EMPTY return BOOLEAN; ...

    procedure EXCHANGE (FROM: in INDEX; TO: in INDEX); ...

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

    procedure RETRIEVE ...
    exception TABLE_FULL ...

begin
    {code for intialization of the table linkages}
end {TABLE_MANAGER};
```

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

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

The details of implementing such packages can be quite complex, in this case involving a two way linked table of internal items. A local housekeeping procedure EXCHANGE is used to move an internal item between the busy and the free lists. The initial table linkages are established by the initialization part.

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

7.6 Scope Rules

A scope denotes a region of text in which an identifier is known with a single meaning. Subprograms, blocks, paths, and definition modules introduce new scopes. Local scopes are also defined by for loops, record types, and variant parts.

An identifier of an outer scope may be redeclared in a given inner declarative part unless it is used in that declarative part or unless it is a type name. The inner redeclaration has the effect of hiding the outer declaration. Overloading, i.e. redeclaration of a subprogram with different parameter types, is possible even within the same declarative part and does not hide previous subprogram definitions (see section 6.5).

The rules defining the meaning of identifiers within a given scope depend on the presence (or absence) of an import clause in the corresponding declarative part.

```
import_clause ::= import none,  |  import [only] item {,item};

item ::= identifier [(renaming_clause {,renaming_clause})]

renaming_clause ::= new_name == old_name
```

8. PARALLEL PROCESSING

This section specifies the constructs for allowing control paths to operate in parallel. The control paths may be implemented on multiple processors or with interleaved execution on a single processor.

8.1 Path Declarations

A path declaration introduces the names of one or more parallel paths and specifies the information needed to communicate with other paths. It may also contain a translation mode specifying whether the path is separately compiled or generic. Communication between paths is handled by associating boxes with each path.

```
path_declaration ::=
    identification: [translation_mode] path box_part;

box_part ::= [(box_definition { ; box_definition })]

box_definition ::=
    identification: box_mode box [type] [interrupt_clause]

box_mode ::= in | out

identification ::=
    identifier [(range_denotation { , range_denotation })]
```

A box may be specified with a type and a mode in or out, which indicates that the box is used for receiving data from other paths or for sending data to other paths. If no type is specified for a box, it is used only for synchronization and must have the mode in. The interrupt clause will be described in section 8.7.

A path or box identification may specify a family of identical paths or boxes, each denoted by one or more indices from a specified range.

Examples:

```
KEYBOARD_DRIVER   : path (LINE: in box LINE_IMAGE);

DECODER           : path (CHAR: in box CHARACTER);

CONTROLLER        : path(START_READ  : in box;
                         START_WRITE : in box;
                         STOP_READ   : in box;
                         STOP_WRITE  : in box);

SCHEDULER         : path(START(1 .. 3) : in box;
                         STOP          : in box);

TERMINAL(1 .. N)  : path(MESSAGE  : in box LINE_IMAGE;
                         RESPONSE : out box LINE_IMAGE);
```

8.2 Path Bodies

A path body describes the execution of a path. The body will generally contain statements that control the execution of other paths. Within the path body, the out boxes of a given type behave like local variables, and the in boxes of a given type behave like unassignable local variables.

```
path_body ::=
    path path_name box_part;
    declarative_part begin statement_list end;
```

Example:

```
path KEYBOARD_DRIVER (LINE: in box LINE_IMAGE);
begin
    loop
        receive LINE;
        for I in 1 . . 80 loop
            connect DECODER (CHAR := LINE(1) );
            exit when LINE(1) = "?";
        repeat;
    repeat;
end [KEYBOARD_DRIVER];
```

8.3 Synchronization Statements

Synchronization statements specify the communication between paths.

```
synchronization_statement ::= do_inner_paths_statement
    | local_request_statement | connect_statement
```

A do inner paths statement initiates execution of one or more paths. A local path request specifies that a local box in the path is ready for connection from another path. A connect statement specifies that the path is ready to connect with a box in another path.

A rendezvous is achieved when a path is ready for a connection to one of its local boxes and when another path is ready to connect to the same box. Whenever a rendezvous occurs, any specified data transfer takes place, and both the local path and the connecting path continue execution.

8.3.1 Initiating paths

The execution of paths is initiated with a do inner paths statements.

```
do_inner_paths_statement ::= inner;
```

When control reaches the statement, the path declarations of the local declarative part are elaborated, and all declared paths may begin execution. Execution at the inner statement is suspended until all inner paths have completed their execution.

8.3.2 Local request statements

A local path request allows for local synchronization and possible data transmission.

```
local_request_statement ::= local_request_clause;

local_request_clause ::= send box_denotation
    | receive box_denotation | delay expression

box_denotation ::= box_name[(expression{ ,expression })]
```

A send or receive request must belong to the statement list of the path owning the denoted boxes. A receive request can only be associated with an **in** box; a send request can only be associated with an **out** box.

For a box denotation specifying a family of boxes, the expressions denote index values for one member of the family. Delay requests will be described in section 8.6.

8.3.3 Connect statements

A connect statement specifies that a path is ready to connect with a box in another path.

```
connect_statement ::=
        connect path_denotation (box_denotation [:= expression] );
    |   connect path_denotation (box_denotation [=: variable] );

path_denotation ::= path_name [(expression{ ,expression })]
```

An expression or variable is given after a box denotation to denote a transfer of data. The type of the expression or variable must be identical to that of the box denotation. The specified box must belong to the specified path.

For a path denotation specifying a family of paths, the expressions specify the index values for one member of the path family.

8.3.4 Rendezvous of local requests with connect statements

There are three rendezvous possibilities:

(1) If the box has not been declared with a type (and thus has the mode in), the local request must be a receive request and the corresponding connect statement must only specify the box.

(2) If the box has been declared with a type and has the mode in, the local request must be a receive request and the corresponding connect statement must specify an expression value for assignment to the box.

(3) If the box has been declared with a type and has the mode out, the local request must be a send request and the corresponding connect statement must specify a variable to which the value in the box is assigned.

In each case, a rendezvous results in a synchronization of path execution. In the second and third cases, a rendezvous also results in the transfer of data.

If a send or receive request is issued before a corresponding connect statement, execution of the sending or receiving path is suspended until a corresponding connect statement is issued. Similarly, if a connect statement is issued before a corresponding send or receive request, the connecting path is suspended.

There may be several connecting paths waiting for rendezvous on a given box. In this case the first issued connect statement is used for the rendezvous. The remaining connect statements will be processed on a first in, first out basis by subsequent local requests on the same box.

Examples of send or receive requests:

```
(1) receive STOP;

(2) receive LINE;

(3) send RESPONSE;
```

Examples of associated connect statements in other paths:

```
(1) connect SCHEDULER (STOP);

(2) connect KEYBOARD_DRIVER (LINE := NEW_LINE);

(3) connect TERMINAL(1) (RESPONSE := ANSWER);
```

8.4 Select Statements

A select statement allows a choice of one or more statement lists based on their corresponding when conditions and local request clauses.

```
select_statement ::=
    select selected_box {,selected_box} of
        when [condition] local_request_clause => statement_list {}
    end select;

selected_box ::= box_denotation | multiple_box_selection
```

All boxes mentioned in the list of selected boxes must be different. Each of these boxes must appear in exactly one of the local request clauses. The conditions must only contain variables that are local to the path.

A local request clause is said to be open if its corresponding when condition is true, and closed otherwise. A local request clause with no condition is considered open.

Execution of a select statement proceeds as follows:

(a) All when conditions are evaluated to determine which local request clauses are open.

(b) If there are open clauses, the select statement achieves a rendezvous when any one of its local request clauses is matched by a connect statement issued from another path. When a rendezvous is achieved, the statement list associated with the corresponding local request clause is executed.

(c) If there are no open clauses, the select statement has no effect.

In general, several local path boxes may have been connected before a select statement is encountered. As a result several local request clauses may be matched with connects. In this case, execution of a select statement results in (non-deterministically) executing any one of the matched select options.

Example:

```
path CONTROLLER (START_READ  : in box,
                 START_WRITE : in box,
                 STOP_READ   : in box;
                 STOP_WRITE  : in box);
    READERS: INTEGER := 0;
begin
    loop
        select START_READ | START_WRITE | STOP_READ of
            when READERS = 0 receive START_WRITE =>
                receive STOP_WRITE;
            when receive START_READ =>
                READERS := READERS + 1;
            when receive STOP_READ =>
                READERS := READERS - 1;
        end select;
    repeat;
end;
```

8.4.1 Multiple box selections

A multiple box selection specifies a range of boxes in a given indexed box family.

```
multiple_box_selection ::=
    box_name (box_index { ,box_index})

box_index ::= all identifier in range_denotation
```

An identifier of a multiple box selection may only appear in the when clause whose local request clause mentions the box family name. This when clause has the same effect as the set of when clauses obtained by substitution of all values of the range to the identifier. For example the select statement

```
select B (all I in U . . V) of
    when C(I) receive B(I)=> S(I);
end select;
```

has the same meaning as the following expanded select statement:

```
select B(U) ! . . . ! (B(V) of
    when C(U) receive B(U) => S(U);
    . . .
    when C(V) receive B(V) => S(V);
end select;
```

8.5 Example of Parallel Processing

The following example defines a buffering path to smooth variations in the speed of output between a producing process and the speed of input to some consuming process. For instance, the path for the producing process may contain:

```
loop
    GET (CHAR, DEVICE := UNIT_A);
    connect BUFFERING (IN_CHAR := CHAR);
    exit when (CHAR = END_OF_TRANSMISSION);
repeat;
```

and the path for the consuming process may contain:

```
loop
    connect BUFFERING (FETCH);
    connect BUFFERING (OUT_CHAR := CHAR);
    PUT (CHAR, DEVICE := UNIT_B);
    exit when (CHAR = END_OF_TRANSMISSION);
repeat;
```

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

Example:

```
path BUFFERING (FETCH    : in box;
                IN_CHAR  : in box CHARACTER;
                OUT_CHAR : out box CHARACTER);

    BUFFER_SIZE: constant = 100;
    type BUFFER_INDEX = (1 . . BUFFER_SIZE);
    BUFFER: array (BUFFER_INDEX) of CHARACTER;

    IN_INDEX, OUT_INDEX: BUFFER_INDEX := 1;
    COUNT: (0 . . BUFFER_SIZE) := 0;

begin
    loop
        select IN_CHAR ! FETCH of
            when COUNT < BUFFER_SIZE receive IN_CHAR =>
                BUFFER(IN_INDEX) := IN_CHAR;
                IN_INDEX := (IN_INDEX mod BUFFER_SIZE) + 1;
                COUNT    := COUNT + 1;
            when COUNT > 0 receive FETCH =>
                OUT_CHAR := BUFFER(OUT_INDEX);
                OUT_INDEX := (OUT_INDEX mod BUFFER_SIZE) + 1;
                COUNT    := COUNT - 1;
                send OUT_CHAR;
        end select;
    repeat;
end {BUFFERING};
```

8.6 Delay Requests

A delay clause is implicitly associated with a "box" linked to the system clock. The expression in the delay clause specifies an interval of time, after which an implicit path issues a connect statement to the "box". As a result, the path with the delay clause is suspended for the designated time interval.

For select statements with a delay clause, another box in the list of select options may be connected during the delay interval. If this occurs, the delay is cancelled and the connected select option is processed.

All real time values may be expressed in terms of translation time constants defined in a standard library module for the particular object machine. These values are given in the basic time unit of the clock. Such constants will include declarations for HR (for hours), MN (for minutes), and SEC (for seconds). If the pragmat SIMULATION has been specified, then time is managed by a simulated time clock.

Example:

```
path REFRESH_POSITION;
    import only POSITION, SPEED;
    LAST_TIME, THIS_TIME: INTEGER;
begin
    LAST_TIME  := SYSTEM'CLOCK;
    POSITION   := 0;
    loop
        delay 10*SEC;
        THIS_TIME  := SYSTEM'CLOCK;
        POSITION   := POSITION + SPEED*(THIS_TIME - LAST_TIME);
        LAST_TIME  := THIS_TIME;
    repeat;
end;
```

8.7 Interrupts

A box definition may have an interrupt clause specifying an expression giving a particular interrupt level.

```
interrupt_clause ::= interrupt expression
```

Interrupt boxes must have the mode in, with no associated type. An interrupt box is implicitly linked to a hardware interrupt, whose name becomes that of the box.

An occurrence of a hardware interrupt acts as a connect statement to the interrupt box of a path. As such, a receive request with an interrupt box results in a suspension of the path until the interrupt occurs. An interrupt linked with a box is automatically unmasked when the path executes a receive request on the box.

In a select statement containing a local request for an interrupt box, the interrupt is masked if another box is connected before the interrupt occurs.

Example:

```
path CARD_READER_INTERRUPT (ATTENTION : in box;
                            DONE: in box interrupt 4);
begin
    loop
        receive ATTENTION;
        select
            when receive DONE => connect CR_DRIVER(FINISH);
            when delay 2*SEC  => connect CR_DRIVER(EMPTY);
        end select;
    repeat;
end;
```

8.8 Path Attributes and Predefined Path Functions

There are several attributes and predefined functions associated with paths.

For a path P, the attribute P'PRIORITY gives the path priority. The predefined procedure SET_PRIORITY takes an integer argument and sets the priority of a path to the integer value. Before such a call, all paths have the same standard priority.

When a path of a family F needs to reference its own index, for example to pass it to another path, it may use the attribute F'INDEX for that purpose. Similarly F''INDEX, may be used for the second index of a doubly indexed family, and so forth.

For a box B, the attribute B'COUNT gives the number of external paths that have issued a connect statement to the box but have not yet been serviced.

For a path P, the attribute P'CLOCK gives the cumulative processing time on the path. The real time system clock may be accessed with the attribute SYSTEM'CLOCK, where SYSTEM is a predefined name denoting the system. The path execution clock may be set to zero with the predefined procedure RESET_CLOCK of no arguments.

8.9 Scheduling of Multiple Paths

There may be multiple paths that are ready to be executed by the system processors. In choosing the paths to be processed, paths with the highest priority are processed first. Paths of the same priority level are processed on a first in, first out basis.

8.10 Low Level Input-Output Operations

A low level input-output operation is an operation acting on a physical file. Such an operation is handled by using one of the two predefined procedures SEND_CONTROL and RECEIVE_CONTROL.

The SEND_CONTROL procedure may be used to send control information to a given physical file. Sending control information to a physical file may result in starting a data transfer.

The RECEIVE_CONTROL procedure may be used to monitor the execution of an input-output operation by requesting monitor information from the physical file.

For such operations the kind and formats of required control information will depend on the machine and physical file characteristics. Hence these procedures will be predefined operations declared in a standard definition module for a given implementation.

Example:

```
type DEVICE_ADDRESS  = INTEGER;
type I_O_RANGE       = (0 .. 31);
type I_O_STATUS      = SET(I_O_RANGE);

RECEIVE_CONTROL: procedure (DEVICE: in DEVICE_ADDRESS;
                            RESULT: out I_O_STATUS);
```

9. EXCEPTION HANDLING

This section defines the facilities for dealing with exceptional situations that cause a suspension of normal program execution.

The environments whose execution can be prematurely terminated by an exception are blocks, subprograms, and paths. Exceptions are introduced by exception declarations. Exception handlers are subprogram bodies to which control is passed when an exception occurs.

9.1 Exception Declarations

An exception is declared as a subprogram whose subprogram nature is **exception**. The declaration identifies the scope in which the exception may be raised and processed by a corresponding handler. An exception subprogram may only have in parameters.

Several exceptions are predefined in the standard prelude. These include

OVERFLOW	For exceeding the maximum allowed value of a number
ZERO_DIVIDE	For dividing a number by zero
RANGE_ERROR	For exceeding the declared range of a variable
ILLEGAL_DATA	For a data type error on input
INVALID_ASSERTION	For violating an assertion
UNINITIALIZED	For accessing the value of an uninitialized variable
TERMINATE	For prematurely terminating the execution of a path or a subprogram
OTHER_EXCEPTIONS	For dealing with any exception for which no explicit handler is given in the current scope.

Examples of exception declarations:

```
SINGULAR          : exception;
END_OF_FILE       : exception;

CANCEL_REQUEST    : exception (CODE: REQUEST_CODE);
STORAGE_OVERFLOW  : exception (ZONE, SIZE: INTEGER);
```

9.2 Exception Handlers

The processing of an exception is specified by giving a subprogram body for handling the exception. An exception may be processed by different handlers in different scopes. Specifically, in any block, subprogram, or path that is within the scope of an exception declaration, a local body may be provided to handle the corresponding exception.

When a handler is invoked following the corresponding exception, execution of the handler replaces the remainder of the execution of the block, subprogram, or path where the handler is provided.

Since the handler acts as a substitute for the corresponding unit, any statement that is legal within the unit may be used within the handler. For example, a handler within a function has access to its parameters and may issue a return statement on behalf of the function.

Examples:

```
exception SINGULAR;
begin
    PRINT ("MATRIX IS SINGULAR");
end;

exception CANCEL_REQUEST (CODE: REQUEST_CODE);
begin
    if CODE = ABORT then
        raise TERMINATE;
    else
        DISPLAY ("REQUEST CANCELLED");
    end if;
end;
```

9.3 Raising Exceptions

An exception is implicitly raised when an operation leads to a predefined exception situation, or is explicitly raised by an exception statement.

```
exception_statement : :=
    raise [subprogram_call] [for path_name];
```

An exception statement raises the exception named by the subprogram call. The call may specify input parameters to be passed to the corresponding handler. If no subprogram call is given, a raise statement reraises the most recently raised exception. An exception statement can also be used to raise an exception in another path.

Examples:

```
raise;
raise SINGULAR;
raise CANCEL_REQUEST (CODE := CURRENT_REQUEST_CODE);
raise TERMINATE for PRINTER;
```

9.3.1 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 the suspended block or subprogram does not contain a local handler for the exception, execution of the block or subprogram is terminated and the same exception is reraised in the outer block or in the calling subprogram.

(b) If a local handler has been provided, execution of the handler replaces execution of the remainder of the block or subprogram.

For example, consider the following program structure.

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

    procedure Q;
        exception ERROR; [handler E2]
        begin
            ...
        end;
    begin
        R;
        ... [exception possibility (2) ]
    end;

    procedure R;
    begin
        [exception possibility (3) ]
    end;

    exception ERROR; [handler E1]
    begin
        ...
    end;

begin [P]
        ... [exception possibility (1) ]
    Q;
        ...
end [P];
```

The following cases may arise

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

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

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

Note that case (3) results in a dynamic binding, since the exception in R resuits in passing control to a local subprogram of Q that is not visible in R.

Note also that if a local handler were given within R for the predefined exception OTHER_EXCEPTIONS, case (3) above would cause execution of the handler for OTHER_EXCEPTIONS rather than direct termination of R.

9.3.2 Raising exceptions in other paths

A path can raise an exception in another named path.

If the other path is active, its execution is suspended and a handler may be dynamically invoked as described above. If the other path is not currently active, processing of the exception is deferred until the path becomes active.

If no handler exists in the other path, the same exception will be reraised in the outer subprogram or path at the rejoin point after the inner statement that initiated the path.

9.4 Suppressing Exceptions

Exception conditions may be suppressed within a given scope by including in its declarative part a pragmat of the form:

```
pragmat SUPPRESS (identifier_list)
```

Each designated exception is suppressed within the scope. As a result, no checks are provided to insure that the exceptions do not arise. This facility may be especially useful for the predefined exceptions, since detection of some of them may be expensive unless aided by special hardware. Should they arise, the results may be unpredictable.

Examples:

```
pragmat SUPPRESS (RANGE_ERROR,SUSCRIPT_ERROR)
pragmat SUPPRESS (INVALID_ASSERTION)
```

10. REPRESENTATION SPECIFICATIONS

Representation specifications specify the mapping between data types and features of the underlying machine that execute programs. Representation specifications may be more or less direct: in some cases they completely specify the mapping, in other cases they provide criteria for choosing a mapping.

Representation specifications must appear immediately after the declaration list of a declarative part, and may only be applied to types declared in the same declarative part. If present, they apply to all objects of the type. In the absence of an explicit specification for a given type, the representation will be determined by the translator.

```
representation_specification    : : =
        packing_specification
    |   length_specification
    |   enumeration_type_representation
    |   record_type_representation
```

10.1 Packing Specifications

A packing specification indicates that storage minimization should be the main criterion for selecting the representation of a record or array type. In the absence of a specification, the translator will generally minimize access time to record components or array elements.

```
packing_specification    : : =
    for type_name use packing;
```

For array types, packing specifications are allowed only if the element type is not itself a composite type.

Examples:

```
for MATRIX use packing;
for FILE_DESCRIPTOR use packing;
```

10.2 Length Specifications

There are three forms of length specifications. All forms include an expression whose value is expressed in bits. Attributes may often be used to simplify the writing of these expressions.

```
length specification    : : =
        for type_name use expression;
    |   for path_name use expression;
    |   for access_type_name use expression;
```

The first form indicates that objects of the type should be represented with a specified number of bits. This number must be known at translation time, and must be at least equal to the minimum needed for the representation of variables of the type. A length specification may be used to achieve a biased representation.

Examples:

```
type BIASED = INTEGER range (10_000 .. 10_255);

for COLOR     use ONE_BYTE;
for ELEMENT   use INTEGER'SIZE;
for BIASED    use 8;
```

The second form of length specification may be used to indicate the amount of stack space to be allocated to a given path. This amount must be known at translation time.

Example: [the constant PAGE is expressed in bits]

```
for PRINTER use 4*PAGE;
```

The last form of length specification is used to specify the amount of stack space to be reserved for the collection of dynamic records designated by variables of an access type.

Example: [a collection of 2000 persons]

```
for PERSON use 2000 * PERSON'RECORD_SIZE;
```

10.3 Enumeration Type Representations

An enumeration type representation specifies the internal codes for the elements of an enumeration type.

```
enumeration_type_representation    : : =
    for type_name use array_aggregate;
```

The array aggregate used to specify this mapping is an array aggregate of type

```
array (type_name) of INTEGER
```

All enumeration values must be provided with distinct integer values, and these values must be known at translation time. The integer values specified for the representation of an ordered enumeration type must satisfy the order relation of the type.

An actual array parameter whose index is an enumeration type with a non-contiguous representation cannot be associated with a formal array parameter whose index is specified by *.

Example:

```
type MIX_CODE = (ADD ! SUB ! MUL ! LDA ! STA ! STZ);

for MIX_CODE use
    (ADD:1, SUB:2, MUL:3, LDA:8, STA:24, STZ:33);
```

10.4 Record Type Representations

A record type representation specifies the storage representation of records, i.e., the order, position, and size of record components.

```
record_type_representation    : : =
    for type_name use record [alignment expression;]
        {component_representation }end record;

    component_representation    : : =
        variable_name at expression bits range_denotation;
```

An alignment clause specifies that records must be allocated at addresses that are exact multiples of the number of bits specified.

The position of a component is given by the position of the storage unit containing the first bit of the component (**at** clause) and a bit range (**bits** clause).

For a given machine, the size of a storage unit is defined by the configuration dependent constant SYSTEM'STORAGE_UNIT. The first storage unit of a record is numbered 0. The first bit of a storage unit is numbered 0. The ordering of bits in a storage unit is implementation defined.

A component representation must define a storage field large enough for the component. An implementation may place restrictions on how fields overlap storage boundaries.

All expressions appearing in a record type representation must have values that are known at translation time. Translators must check that record components of a given variant do not overlap.

A component representation may also be used to specify the address and width of a variable.

Example:

```
type PROGRAM_STATUS_WORD =
    record
        SYSTEM_MASK              :   array (0 .. 7) of BOOLEAN;
        PROTECTION_KEY           :   (0 .. 15);
        MACHINE_STATE            :   array (A ! M ! W ! P) of BOOLEAN;
        INTERRUPT_CAUSE          :   INTERRUPTION_CODE;
        ILC    :   (0 .. 3);
        CC     :   (0 .. 3);
        PROGRAM_MASK    :   array (FIX ! DEC ! EXP ! SIGNIF) of BOOLEAN;
        INST_ADDRESS: ADDRESS;
    end record;

for PROGRAM_STATUS_WORD use
    record alignment 64;
        SYSTEM_MASK       at 0 bits 0 .. 7;
        PROTECTION_KEY    at 0 bits 8 .. 11;
        MACHINE_STATE     at 0 bits 12 .. 15;
        INTERRUPT_CODE    at 0 bits 16 .. 31;
        ILC  at 1 bits 0 .. 1;
        CC   at 1 bits 2 .. 3;
        PROGRAM_MASK at 1 bits 4 .. 7;
        INST_ADDRESS   at 1 bits 8 .. 31;
    end record;
```

10.5 Change of Representations

Only one representation can be defined for a given type. As a consequence if an alternate representation of a given type is desired, it is necessary to define a second type which is logically equivalent (has the same properties) but has a different representation.

Example:

```
-- PACKED_DESCRIPTOR and DESCRIPTOR are two different
-- types with identical properties

type DESCRIPTOR =
    record
        ...
    end;

type PACKED_DESCRIPTOR = DESCRIPTOR;

for PACKED_DESCRIPTOR use packing;
```

Change of representations may be accomplished by assignment with explicit type conversions. Such conversions are legal for types declared as logically equivalent.

Examples:

```
D   :   DESCRIPTOR;
P   :   PACKED_DESCRIPTOR;

P   := PACKED_DESCRIPTOR (D);    -- pack
D   := DESCRIPTOR (P);           -- unpack
```

10.6 Configuration and Machine Dependent Constants

Configuration dependent constants are expressed as attributes of the predefined name SYSTEM. Similarly, translator options may be interrogated with boolean attributes of the predefined name OPTION. Other implementation dependent properties of specific program constructs may be interrogated using appropriate attribute qualifiers.

Examples:

```
SYSTEM'STORAGE_UNIT          OPTION'SPACE
SYSTEM'MEMORY_SIZE           OPTION'TIME
SYSTEM'NAME                  OPTION'LIST

REAL'RADIX                   SMALLINT'IMPLEMENTED_RANGE
INTEGER'SIZE                 TABLE'ADDRESS

X. COMPONENT'POSITION        [position of COMPONENT in storage units]
X. COMPONENT'FIRST_BIT       [first bit of bit range]
X. COMPONENT'LAST_BIT        [last bit of bit range]
```

11. OVERALL PROGRAM STRUCTURE AND COMPILATION ISSUES

This section describes the overall structure of programs and the facilities for separate compilation. In general, a program is a collection of one or more compilation units, which are subprograms, definition modules, or paths. Exception bodies may not be separately compiled, since their use is local to a given scope. Compilation units may be grouped in libraries to be reused by several different programs.

This section also describes generic program units, the facilities available for conditional compilation, and the treatment of configuration dependent features.

11.1 Compilation Units

The body of a subprogram, path or definition module declared with the translation mode **separate** is called a compilation unit and is compiled separately. This means that its text may be submitted to the translator separately from the rest of the program text.

Algorithm modules, another form of compilation units will be described later.

```
compilation_unit : : = body     |    algorithm_module
```

Declarations of separately compiled units may only appear within the outermost declarative part of a subprogram, which itself is separately compiled. The main program is implicitly declared in the standard prelude as

```
MAIN :  separate procedure;
```

Examples of separately compiled units:

```
procedure MAIN;
    L   :   constant = 100;
    D   :   separate definition;
    A   :   separate procedure (X: in INTEGER);
begin
    ...
end (MAIN);

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

definition D;
    LIMIT   :   constant INTEGER = 1000;
    TABLE   :   array (1 . . LIMIT) of INTEGER;
end (D);

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

```
procedure A (X: in INTEGER);
    import D;
    Y   :   INTEGER;
    C   :   separate procedure;
begin
    ...
end (A);
```

11.2 Recompilations and Scope Rules

The scope rules applicable to separate subprograms and paths are identical to those of normal subprograms and paths. For example, a separate subprogram C declared in a subprogram A has access to the identifiers declared in A, exactly as if C were declared as a normal procedure.

The rules of recompilations follow the scope rules: a compilation unit must be recompiled whenever another unit which it sees is recompiled, since the visible information may have been changed. As a consequence, recompilation of the enclosing unit A requires a recompilation of inner separate subprograms or paths like C.

In addition, if a compilation unit C imports a definition D, it must be recompiled whenever D is recompiled.

A different rule applies to separate definition modules. A separate definition module does not have access to the local entities of the procedure where it is declared, unless it explicitly imports the name of the procedure. As a consequence, a separate definition must only be recompiled when any of the units it imports is recompiled.

The declaration of a separate definition module plays the same role as that of a normal definition module; it identifies the point where the definition must be instantiated. A separate definition may be compiled before the procedure where it is declared, provided it does not import the name of the procedure. This possibility is essential for library definition modules.

In the previous example:

(a) Within procedure A, the identifiers L and D declared in MAIN are visible. The identifiers LIMIT and TABLE imported from the definition module D are also visible.

(b) Within definition D, the identifiers L and A declared in MAIN are not visible since D does not import MAIN. Thus D may be compiled either before or after MAIN.

(c) Recompilation of MAIN requires recompilation of A

(d) Recompilation of D requires recompilation of A

11.3 Algorithm Modules

Changes in a subprogram body or within the algorithm part of a definition module do not affect units that import the definition, since these units only have access to the visible part.

As a result, to minimize recompilations, a definition module may be compiled in two units: a definition module containing only its visible and private declarative parts, and an algorithm module containing its algorithm and initialization parts.

Similarly a subprogram may be compiled in two units: a subprogram body containing only the declarations needed by inner compilation units, and an algorithm module containing local declarations and the statement list of the subprogram.

An algorithm module bears the same name as the subprogram or definition module of which it is a part. Recompilation of the algorithm module of a separate definition does not necessitate recompilation of units importing the definition. Similarly, recompilation of the algorithm module of a separate subprogram does not necessitate recompilation of separate procedures and paths that are declared within the subprogram.

```
algorithm_module : : =
    algorithm identifier; declarative_part
    [begin statement_list] end;
```

For example, consider the definition module RATIONAL_NUMBERS of Section 7.3 declared as a separately compiled unit:

```
RATIONAL_NUMBERS : separate definition;
```

The first compilation unit contains all the information needed by other program units that import the module.

```
definition RATIONAL_NUMBERS;
    type RATIONAL =
        record
            NUMERATOR      :   INTEGER;
            DENOMINATOR    :   (1..INTEGER'LAST);
        end record;
    EQUIV   : function (X,Y: RATIONAL) return BOOLEAN;
    ADD     : function (X,Y: RATIONAL) return RATIONAL;
    MULT    : function (X,Y: RATIONAL) return RATIONAL;
end;
```

Note that the above unit does not contain the function bodies. These are in the separate algorithm module:

```
algorithm RATIONAL_NUMBERS;
    procedure SAME_DENOMINATOR(X,Y: access RATIONAL);
    begin
        [reduces X and Y to common denominator]
    end;

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

    function ADD . . .
    function MULT . . .
end;
```

11.4 Libraries

Libraries can be constructed with separately compiled subprograms, definition modules, and algorithm modules. For standard user packages, such as an application level input-output package, splitting of definition modules into two parts (one corresponding to the user interface, the other containing the bodies) should be systematically used.

11.5 Compilation file

Compilers must preserve the same degree of type safety for separately compiled units as for other units. Consequently, a compilation file containing information on previously compiled units must be maintained by the translator. This information includes symbol tables and information pertaining to the order of previous compilations.

A normal submission to the translator will include the text of the compilation unit and the compilation file. The latter is used for checks and may be updated by the current compilation.

11.6 Conditional Compilation

Statements appearing in a case statement will not be compiled if the case discriminant is known at translation time and if they are not in the alternative selected by the discriminant value.

Similarly an if statement with conditions known at translation time may be used to achieve conditional compilation.

Conditional compilation of declarations may be achieved in a similar fashion with a variant part whose discriminant is known at translation time. Only the declarations of the translation time chosen variant will be compiled. A variant part that appears in contexts other than a record declaration can only be used for conditional compilation; its discriminant must always be a translation time constant.

Variant parts and translation time case statements can be used for compiling program portions that depend on the object machine configuration. In such cases the discriminant will be a translation time constant relating to the configuration.

In the example below, variant parts and translation time case statements are used to produce two alternative programs that differ only on the value of a constant CHOICE, which is set to one of two values before compilation.

Example of conditional compilation:

```
CHOICE: constant (A | B) := A;

procedure ALTERNATE (X : REAL);
    case CHOICE of
        when A => U : REAL;
        when B => V : LONG_REAL;
    end case;
begin
    case CHOICE of
        when A => U := X;
        when B => V := LONG_REAL(X);
    end case;
    [remaining statements of ALTERNATE]
end;
```

Example of resulting choice:

```
procedure ALTERNATE (X: REAL);
    U: REAL;
begin
    U := X;
    [remaining statement of ALTERNATE]
end;
```

11.7 Generic Program Units

Generic program units are program units with translation time parameters specified by a generic clause. Instances of generic program units are declared by generic instantiations.

```
generic_clause : : = generic [ ( identifier_list) ]

generic_instantiation : : =
    identifier: new generic_nature generic_name
    [ (generic_association {, generic_association }) ];

generic_nature : : = function | procedure | definition | path

generic_association : : =
        generic_parameter_name == expression
    |   generic_parameter_name == name
```

The identifier list of the generic clause defines the generic parameters. They may appear anywhere in the body of the generic program unit.

In order to create an instance of a generic program unit, replacements for the generic parameters must be provided by generic associations. Either a translation time expression or the name of a previously declared entity may be substituted for a generic parameter. This substitution is performed in the text of the generic program unit for each generic instantiation.

Instances of generic program units may be used as program units. Note that when several instances of the same generic definition module are imported in the same scope, renaming clauses are usually necessary to resolve name conflicts. Within a generic program unit, the type of a variable V may be denoted by the attribute V'TYPE.

Generic program units may not be separately compiled.

Examples of generic declarations:

```
STACK : generic (ELEM, SIZE) definition;
SWAP  : generic (ELEM) procedure (U,V : access ELEM);
```

Examples of generic instantiations:

```
STACK_INT  : new definition STACK(ELEM == INTEGER, SIZE == 200);
STACK_BOOL : new definition STACK(ELEM == BOOLEAN, SIZE == 100);

EXCHANGE : new procedure SWAP(ELEM == INTEGER);
EXCHANGE : new procedure SWAP(ELEM == REAL);
```

Example of a generic definition module:

```
definition STACK;
    PUSH  :  procedure (E: in ELEM);
    POP   :  procedure (E: out (ELEM);
    STACK_ERROR  :  exception (MODE:  (OVER | UNDER) );

algorithm
    SPACE  :  array (1 .. SIZE) of ELEM;
    INDEX  :  (0 .. SIZE) :=  0;

    procedure PUSH (E: in ELEM);
    begin
        if INDEX = SIZE then
            raise STACK_ERROR(OVER);
        end if;
        INDEX := INDEX + 1;
        SPACE(INDEX) := E;
    end [PUSH];

    procedure POP (E: out ELEM);
    begin
        if INDEX = 0 then
            raise STACK_ERROR(UNDER);
        end if;
        E := SPACE(INDEX);
        INDEX := INDEX - 1;
    end [POP];

    exception STACK_ERROR;
    begin
        raise TERMINATE;
    end [STACK_ERROR];

end [STACK].
```

APPENDIX A. SAMPLE INPUT OUTPUT DEFINITIONS

The general facilities offered by the Green language enable the construction of application level input output facilities without additional language constructs. Thus standard application level packages may be developed for major application classes, current or future, without affecting the core language as seen by the users.

The two examples provided below are meant to indicate the principle of construction of such packages and to illustrate the general form of user interfaces.

The first example, CHARACTER_FILE_HANDLING corresponds to the treatment of character files. The type FILE itself is private (i.e. its name is known but its properties are not directly accessible to the user). The visible part of the definition module declares procedures for standard file operations (creating, opening, closing) and for the procedures GET and PUT. The algorithm part contains the bodies of these procedures. It may also contain local procedures and paths performing the necessary low-level input output operations. The initialization part creates and opens the standard files INPUT and OUTPUT. Notice the use made of named and optional parameters in these calls of CREATE.

```
definition CHARACTER_FILE_HANDLING;

    type FILE = private;

    INPUT, OUTPUT : FILE;

    type FILE_ACCESS_METHOD = (SEQUENTIAL ! INDEXED ! DIRECT);
    type FILE_RECORD_FORMAT = (FIXED ! VARIABLE);
    type ACCESS_RIGHT       = (INPUT_MODE ! OUTPUT_MODE ! UPDATE_MODE);

    CREATE : procedure (NAME          : array (*) of CHARACTER;
                        RECORD_LENGTH : INTEGER := 80;
                        KEY_LENGTH    : INTEGER := 0;
                        BLOCK_LENGTH  : INTEGER := 320;
                        ACCESS_METHOD : FILE_ACCESS_METHOD := SEQUENTIAL,
                        RECORD_FORMAT : FILE_RECORD_FORMAT := FIXED,
                        F             : out FILE);

    DELETE : procedure (F : FILE);
    OPEN   : procedure (F : FILE; A : ACCESS_RIGHT),
    CLOSE  : procedure (F : FILE);

    GET    : procedure (F : FILE; BUF : access array (*) of CHARACTER);
    PUT    : procedure (F : FILE; BUF : access array (*) of CHARACTER);

    {other file handling procedures}
```

```
private
    type FILE = [ . . . ];

algorithm

    [internal tables and bodies of above procedures]

begin

    CREATE(NAME := "SYSIN"; F =: INPUT);
    OPEN(INPUT, INPUT_MODE);

    CREATE(NAME := "SYSOUT"; F =: OUTPUT);
    OPEN(OUTPUT, OUTPUT_MODE);

end [CHARACTER_FILE_HANDLING];
```

The second example is an interface definition module for Pascal-like Input-Output. The corresponding algorithm module may be compiled separately and need not be shown to the user.

```
definition PASCAL_I_O;
    import FILE_CHARACTER_HANDLING;

    LINE_LENGTH: constant = 120;
    subtype FIELD_WIDTH = INTEGER range (0 .. LINE_LENGTH);

    READ:  procedure (V: out CHARACTER;  F : in FILE := INPUT);
    READ:  procedure (V: out INTEGER;    F : in FILE := INPUT);
    READ:  procedure (V: out REAL;       F : in FILE := INPUT);

    WRITE: procedure (E: in  CHARACTER;
                      W: in  FIELD_WIDTH := 1;
                      F: in  FILE := OUTPUT);

    WRITE: procedure (E: in  BOOLEAN;
                      W: in  FIELD_WIDTH := 5;
                      F: in  FILE := OUTPUT);

    WRITE: procedure (E: in  INTEGER;
                      W: in  FIELD_WIDTH := 10;
                      F: in  FILE := OUTPUT);

    WRITE: procedure (E: in  REAL;
                      W: in  FIELD_WIDTH := 20;
                      W1: in FIELD_WIDTH := 0;
                      F: in  FILE := OUTPUT);

    WRITE: procedure (E: in  array (*) of CHARACTER;
                      F: in  FILE := OUTPUT);

end [PASCAL_I_O];
```

Here again the use of optional parameters, and also of overloading permits the user to formulate calls to these procedures in a way which is very similar to Pascal. Thus calls could be written as follows:

```
WRITE(1), WRITE(2E0); WRITE(A = B), WRITE ("THE END");
```

3.9 Declarative parts

```
declarative_part ::= {import_clause} {declaration}
    {representation_specification} {body}

body ::= subprogram_body | definition_module_body | path_body
```

4.1 Variables

```
variable ::=
    variable_name | array_element | slice | record_component

array_element ::= variable (expression {,expression});

slice ::= variable (range_denotation)

record_component ::= variable.component_name | variable.all
```

4.2 Scalar values and attributes

```
scalar_value ::= number | enumeration_value | attribute

attribute ::= denotation'attribute_qualifier

denotation ::=
    name | variable | path_denotation | box_denotation
```

4.3 Expressions

```
expression ::=
      simple_expression [relational_operator simple_expression]
    | simple_expression is [not] range_denotation

simple_expression ::= [simple_expression adding_operator] term

term ::= [term multiplying_operator] factor

factor ::= [unary operator] primary

primary ::= variable | scalar_value | array_aggregate
    | record_aggregate | function_call | (expression)
    | qualified_expression | none
```

4.4 Operators

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

adding_operator ::= + | - | or | xor | &

multiplying_operator ::= * | / | mod | div | and

unary_operator ::= + | - | not | abs
```

4.5 Qualified expressions

```
qualified_expression ::=
    typed_expression | constrained_expression

typed_expression ::= type_name'(expression)

constrained_expression ::= subtype_name'(expression)
```

5. Statements

```
statement_list ::= {[label] statement}

statement ::= simple_statement | compound_statement
    | transfer_statement

simple_statement ::= assignment_statement | allocation_statement
    | subprogram_call_statement | assert_statement
    | synchronization_statement | inline_statement
    | null;

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

transfer_statement ::= loop_exit_statement
    | return_statement | exception_statement
    | goto_statement
label ::= << identifier >>
```

5.1 Assignment statements

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

5.2 Allocation statements

```
allocation_statement ::=
      variable := new record_aggregate
    | variable := new typed_expression;
```

5.3 Subprogram calls

```
subprogram_call_statement ::= subprogram_call;

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

parameter_association ::=
    input_association | output_association | access_association

input_association ::= [formal_parameter =>] expression

output_association ::= [formal_parameter =>] variable

access_association ::= [formal_parameter =>] variable

formal_parameter ::= identifier
```

5.4 Return statements

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

5.5 If statements

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

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

APPENDIX B. SYNTAX SUMMARY

2.9 Lexical elements

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

3.1 Declarations

```
declaration ::=
      element_declaration       | type_declaration
    | subtype_declaration        | access_type_declaration
    | subprogram_declaration     | path_declaration
    | definition_declaration     | generic_instantiation
    | variant_part               | null;
```

3.2 Element declarations

```
element_declaration ::=
      variable_declaration       | renaming_declaration
    | constant_declaration       | deferred_constant_declaration

variable_declaration ::=
    identifier_list: type [:= expression];

renaming_declaration ::=
    identifier: type := variable;

constant_declaration ::=
    identifier: constant [type] := expression;

deferred_constant_declaration ::=
    identifier: constant type;
```

3.3 Type and subtype declarations

```
type_declaration ::= type identifier is type_definition;

type_definition ::= type | private [parameter]

type ::=
      simple_type_definition [constraint]
    | array_type | record_type

simple_type_definition ::= scalar_type | type_denotation

type_denotation ::= type_name | subtype_name | attribute

constraint ::=
    scalar_constraint | array_constraint | record_constraint

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

3.4 Scalar types

```
scalar_type ::= discrete_type | real_type | range

discrete_type ::= scaled_type | enumeration_type

scalar_constraint ::= range (range)

range ::= simple_expression .. simple_expression
```

3.4.2 Scaled types

```
scaled_type ::= scale simple_expression
```

3.4.3 Real types

```
real_type ::= precision simple_expression
```

3.4.4 Enumeration types

```
enumeration_type ::=
      (enumeration_value { ,enumeration_value })
    | (enumeration_value { <enumeration_value })

enumeration_value ::= identifier | character
```

3.5 Array types

```
array_type ::= array (index { ,index }) of type

index ::= range_denotation | *

range_denotation ::= range | type_denotation

array_constraint ::= (range_denotation { ,range_denotation })
```

3.5.2 Array aggregates and strings

```
array_aggregate ::= character_string
    | (type_name) (element_specification { ,element_specification })

element_specification ::= selection => expression

selection ::= selected_value { | selected_value }

selected_value ::=
    number | enumeration_value | range_denotation | others
```

3.6 Record types

```
record_type ::= record component_list end record;

component_list ::= { element_declaration } [variant_part]

variant_part ::= case discriminant of { variant } end case;

discriminant ::= variable_name

variant ::= when selection => component_list
```

3.6.2 Record aggregates and record constraints

```
record_aggregate ::= [type_name'] (component_specification
    { ,component_specification })

component_specification ::=
    component_name { |component_name } => expression

record_constraint ::= record_aggregate
```

3.7 Access types

```
access_type_declaration ::= access type identifier is type;
```

5.6 Case statements

case_statement ::=
  case expression **of** {alternative} **end case**;

alternative ::= **when** selection => statement_list

5.7 Assertion statements

assert_statement ::= **assert** [condition];

5.8 Loop statements

loop_statement ::= [iteration_specification] basic_loop;

basic_loop ::= **loop** statement_list **repeat**

iteration_specification ::=
    while condition | **until** condition
  | **for** loop_parameter **in** [**reverse**] range | denotation

loop_parameter ::= identifier

5.9 Loop exit statements

loop_exit_statement ::= **exit** [**when** condition]

5.10 Blocks

block ::=
  declare declarative_part **begin** statement_list **end**;

5.11 Goto statements

goto_statement ::= **goto** identifier;

6.1 Subprogram declarations

subprogram_declaration ::=
  name: [translation_mode] subprogram_nature formal_part;

subprogram_nature ::= **procedure** | **function** | **exception**

formal_part ::=
  [(parameter_definition {; parameter_definition})] [**return** type]

parameter_definition ::=
  identifier_list: mode type [:= expression]

mode ::= [**in**] | **out** | **access**

name ::= identifier | character_string

translation_mode ::= **separate** | generic_clause

6.3 Subprogram bodies

subprogram_body ::=
  [**inline**] subprogram_nature name formal_part
  declarative_part [**begin** statement_list] **end**

6.6 Code insertions

inline_statement ::= **inline** record_aggregate

7.1 Specification of definition modules

definition_declaration ::=
  identifier: [translation_mode] **definition**;

definition_module_body ::=
  definition identifier; declarative_part
  [**private** declarative_part]
  [**algorithm** declarative_part]
  [**begin** statement_list] **end**;

7.6 Scope rules

import_clause ::= **import none**; | **import** [**only**] item {, item};

item ::= identifier [(renaming_clause {, renaming_clause})]

renaming_clause ::= new_name => old_name

8.1 Path declarations

path_declaration ::=
  identification [translation_mode] **path** box_part;

box_part ::= (box_definition {, box_definition});

box_definition ::=
  identification box_mode **box** [type] [interrupt_clause]

box_mode ::= **in** | **out**

identification ::=
  identifier [(range_denotation {, range_denotation})]

8.2 Path bodies

path_body ::=
  path path_name box_part;
  declarative_part **begin** statement_list **end**;

8.3 Synchronization statements

synchronization_statement ::= do_inner_paths_statement
  | local_request_statement | connect_statement

8.3.1 Initiating paths

do_inner_paths_statement ::= **inner**;

8.3.2 Local request statements

local_request_statement ::= local_request_clause;

local_request_clause ::= **send** box_denotation
  | **receive** box_denotation | **delay** expression

box_denotation ::= box_name [(expression {, expression})]

8.3.3 Connect statements

```
connect_statement ::=
      connect path_denotation (box_denotation [ := expression] ),
    | connect path_denotation (box_denotation [ := variable] ),

path_denotation ::= path_name [ (expression { ,expression }) ]
```

8.4 Select statements

```
select_statement ::=
   select selected_box { | selected_box } of
      { when [condition] : local_request_clause => statement_list }
   end select;

selected_box ::= box_denotation | multiple_box_selection
```

8.4.1 Multiple box selections

```
multiple_box_selection
   box_name (box_index { ,box_index } )

box_index ::= all identifier in range_denotation
```

8.7 Interrupts

```
interrupt_clause ::= interrupt expression
```

9.3 Raising exceptions

```
exception_statement ::=
   raise [subprogram_call] [ for path_name ];
```

10. Representation specifications

```
representation_specification ::=
      packing_specification
    | length_specification
    | enumeration_type_representation
    | record_type_representation
```

10.1 Packing specifications

```
packing_specification
   for type_name use packing;
```

10.2 Length specifications

```
length_specification ::=
      for type_name use expression;
    | for path_name use expression;
    | for access_type_name use expression;
```

10.3 Enumeration type representations

```
enumeration_type_representation ::=
   for type_name use array_aggregate;
```

10.4 Record type representations

```
record_type_representation ::=
   for type_name use record [alignment expression;]
      { component_representation } end record;

component_representation ::=
   variable_name at expression bits range_denotation;
```

11.1 Compilation units

```
compilation_unit ::= body | algorithm_module
```

11.3 Algorithm modules

```
algorithm_module ::=
   algorithm identifier declarative_part
   [begin statement_list] end;
```

11.7 Generic program units

```
generic_clause ::= generic [ (identifier_list) ];

generic_instantiation ::=
   identifier new generic_nature generic_name
   [ (generic_association { ,generic_association }) ];

generic_nature ::= function | procedure | definition | path

generic_association ::=
      generic_parameter_name => expression
    | generic_parameter_name => name
```

APPENDIX C. INDEX