Clause 2

Subprograms and packages

Subprograms define algorithms for computing values or exhibiting behavior. They may be used as computational resources to convert between values of different types, to define the resolution of output values driving a common signal, or to define portions of a process. Packages provide a means of defining these and other resources in a way that allows different design units to share the same declarations.

There are two forms of subprograms: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. Certain functions, designated *pure* functions, return the same value each time they are called with the same values as actual parameters; the remainder, *impure* functions, may return a different value each time they are called, even when multiple calls have the same actual parameter values. In addition, impure functions can update objects outside of their scope and can access a broader class of values than can pure functions. The definition of a subprogram can be given in two parts: a subprogram declaration defining its calling conventions, and a subprogram body defining its execution.

Packages may also be defined in two parts. A package declaration defines the visible contents of a package; a package body provides hidden details. In particular, a package body contains the bodies of any subprograms declared in the package declaration.

2.1 Subprogram declarations

A subprogram declaration declares a procedure or a function, as indicated by the appropriate reserved word.

```
subprogram_declaration ::=
    subprogram_specification ;

subprogram_specification ::=
        procedure designator [ ( formal_parameter_list ) ]
    | [ pure | impure ] function designator [ ( formal_parameter_list ) ]
            return type_mark

designator ::= identifier | operator_symbol

operator_symbol ::= string_literal
```

The specification of a procedure specifies its designator and its formal parameters (if any). The specification of a function specifies its designator, its formal parameters (if any), the subtype of the returned value (the *result subtype*), and whether or not the function is pure. A function is *impure* if its specification contains the reserved word **impure**; otherwise, it is said to be *pure*. A procedure designator is always an identifier. A function designator is either an identifier or an operator symbol. A designator that is an operator symbol is used for the overloading of an operator (see 2.3.1). The sequence of characters represented by an operator symbol must be an operator belonging to one of the classes of operators defined in 7.2. Extra spaces are not allowed in an operator symbol, and the case of letters is not significant.

NOTES

1—All subprograms can be called recursively.

2—The restrictions on pure functions are enforced even when the function appears within a protected type. That is, pure functions whose body appears in the protected type body ~~may~~ must[1] not directly reference variables declared immediately within the declarative region associated with the protected type. However, impure functions and procedures whose bodies appear in the protected type body may make such references. Such references are made only when the referencing subprogram has exclusive access to the declarative region associated with the protected type.

2.1.1 Formal parameters

The formal parameter list in a subprogram specification defines the formal parameters of the subprogram.

formal_parameter_list ::= *parameter*_interface_list

Formal parameters of subprograms may be constants, variables, signals, or files. In the first three cases, the mode of a parameter determines how a given formal parameter ~~may be~~ is[2] accessed within the subprogram. The mode of a formal parameter, together with its class, ~~may also determine~~ also determines[3] how such access is implemented. In the fourth case, that of files, the parameters have no mode.

For those parameters with modes, the only modes that are allowed for formal parameters of a procedure are **in**, **inout**, and **out**. If the mode is **in** and no object class is explicitly specified, **constant** is assumed. If the mode is **inout** or **out**, and no object class is explicitly specified, **variable** is assumed.

For those parameters with modes, the only mode that is allowed for formal parameters of a function is the mode **in** (whether this mode is specified explicitly or implicitly). The object class must be **constant**, **signal**, or **file**. If no object class is explicitly given, **constant** is assumed.

In a subprogram call, the actual designator (see 4.3.2.2) associated with a formal parameter of class **signal** must be a name denoting[4] a signal. The actual designator associated with a formal of class **variable** must be a name denoting[5] a variable. The actual designator associated with a formal of class **constant** must be an expression. The actual designator associated with a formal of class **file** must be a name denoting[6] a file.

NOTE

—Attributes of an actual are never passed into a subprogram: references to an attribute of a formal parameter are legal only if that formal has such an attribute. Such references retrieve the value of the attribute associated with the formal.

2.1.1.1 Constant and variable parameters

For parameters of class **constant** or **variable**, only the values of the actual or formal are transferred into or out of the subprogram call. The manner of such transfers, and the accompanying access privileges that are granted for constant and variable parameters, are described in this subclause.

For a nonforeign subprogram having a parameter of a scalar type or an access type, the parameter is passed by copy. At the start of each call, if the mode is **in** or **inout**, the value of the actual parameter is copied into the associated formal parameter; it is an error if, after applying any conversion function or type conversion present in the actual part of the applicable association element (see 4.3.2.2), the value of the actual parameter does not belong to the subtype denoted by the subtype indication of the formal. After completion of the subprogram body, if the mode is **inout** or **out**, the value of the formal parameter is copied back into the associated actual parameter; it is

1. IR1000.4.7.
2. IR1000.4.7.
3. IR1000.4.7.
4. IR1000.4.1.
5. IR1000.4.1.
6. IR1000.4.1.

similarly an error if, after applying any conversion function or type conversion present in the formal part of the applicable association element, the value of the formal parameter does not belong to the subtype denoted by the subtype indication of the actual.

For a nonforeign subprogram having a parameter whose type is an array or record, an implementation may pass parameter values by copy, as for scalar types. If a parameter of mode **out** is passed by copy, then the range of each index position of the actual parameter is copied in, and likewise for its subelements or slices. Alternatively, an implementation may achieve these effects by reference; that is, by arranging that every use of the formal parameter (to read or update its value) be treated as a use of the associated actual parameter throughout the execution of the subprogram call. The language does not define which of these two mechanisms is to be adopted for parameter passing, nor whether different calls to the same subprogram are to use the same mechanism. The execution of a subprogram is erroneous if its effect depends on which mechanism is selected by the implementation.

For a subprogram having a parameter whose type is a protected type, the parameter is passed by reference. It is an error if the mode of the parameter is other than **inout**.

For a formal parameter of a constrained array subtype of mode **in** or **inout**, it is an error if the value of the associated actual parameter (after application of any conversion function or type conversion present in the actual part) does not contain a matching element for each element of the formal. For a formal parameter whose declaration contains a subtype indication denoting an unconstrained array type, the subtype of the formal in any call to the subprogram is taken from the actual associated with that formal in the call to the subprogram. It is also an error if, in either case, the value of each element of the actual array (after applying any conversion function or type conversion present in the actual part) does not belong to the element subtype of the formal. If the formal parameter is of mode **out** or **inout**, it is also an error if, at the end of the subprogram call, the value of each element of the formal (after applying any conversion function or type conversion present in the formal part) does not belong to the element subtype of the actual.

NOTE

—For parameters of array and record types, the parameter passing rules imply that if no actual parameter of such a type is accessible by more than one path, then the effect of a subprogram call is the same whether or not the implementation uses copying for parameter passing. If, however, there are multiple access paths to such a parameter (for example, if another formal parameter is associated with the same actual parameter), then the value of the formal is undefined after updating the actual other than by updating the formal. A description using such an undefined value is erroneous.

2.1.1.2 Signal parameters

For a formal parameter of class **signal**, references to the signal, the driver of the signal, or both, are passed into the subprogram call.

For a signal parameter of mode **in** or **inout**, the actual signal is associated with the corresponding formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, a reference to the formal signal parameter within an expression is equivalent to a reference to the actual signal.

It is an error if signal-valued attributes 'STABLE, 'QUIET, 'TRANSACTION, and 'DELAYED of formal signal parameters of any mode are read within a subprogram.

A process statement contains a driver for each actual signal associated with a formal signal parameter of mode **out** or **inout** in a subprogram call. Similarly, a subprogram contains a driver for each formal signal parameter of mode **out** or **inout** declared in its subprogram specification.

For a signal parameter of mode **inout** or **out**, the driver of an actual signal is associated with the corresponding driver of the formal signal parameter at the start of each call. Thereafter, during the execution of the subprogram body, an assignment to the driver of a formal signal parameter is equivalent to an assignment to the driver of the actual signal.

If an actual signal is associated with a signal parameter of any mode, the actual must be denoted by a static signal name. It is an error if a conversion function or type conversion appears in either the formal part or the actual part of an association element that associates an actual signal with a formal signal parameter.

If an actual signal is associated with a signal parameter of any mode, and if the type of the formal is a scalar type, then it is an error if the bounds and direction of the subtype denoted by the subtype indication of the formal are not identical to the bounds and direction of the subtype denoted by the subtype indication of the actual.

If an actual signal is associated with a formal signal parameter, and if the formal is of a constrained array subtype, then it is an error if the actual does not contain a matching element for each element of the formal. If an actual signal is associated with a formal signal parameter, and if the subtype denoted by the subtype indication of the declaration of the formal is an unconstrained array type, then the subtype of the formal in any call to the subprogram is taken from the actual associated with that formal in the call to the subprogram. It is also an error if the mode of the formal is **in** or **inout** and if the value of each element of the actual array does not belong to the element subtype of the formal.

A formal signal parameter is a guarded signal if and only if it is associated with an actual signal that is a guarded signal. It is an error if the declaration of a formal signal parameter includes the reserved word **bus** (see 4.3.2).

NOTE

—It is a consequence of the preceding rules that a procedure with an **out** or **inout** signal parameter called by a process does not have to complete in order for any assignments to that signal parameter within the procedure to take effect. Assignments to the driver of a formal signal parameter are equivalent to assignments directly to the actual driver contained in the process calling the procedure.

2.1.1.3 File parameters

For parameters of class **file**, references to the actual file are passed into the subprogram. No particular parameter–passing mechanism is defined by the language, but a reference to the formal parameter must be equivalent to a reference to the actual parameter. It is an error if an association element associates an actual with a formal parameter of a file type and that association element contains a conversion function or type conversion. It is also an error if a formal of a file type is associated with an actual that is not of a file type.

At the beginning of a given subprogram call, a file parameter is open (see 3.4.1) if and only if the actual file object associated with the given parameter in a given subprogram call is also open. Similarly, at the beginning of a given subprogram call, both the access mode of and external file associated with (see 3.4.1) an open file parameter are the same as, respectively, the access mode of and the external file associated with the actual file object associated with the given parameter in the subprogram call.

At the completion of the execution of a given subprogram call, the actual file object associated with a given file parameter is open if and only if the formal parameter is also open. Similarly, at the completion of the execution of a given subprogram call, the access mode of and the external file associated with an open actual file object associated with a given file parameter are the same as, respectively, the access mode of and the external file associated with the associated formal parameter.

2.2 Subprogram bodies

A subprogram body specifies the execution of a subprogram.

```
subprogram_body ::=
    subprogram_specification is
        subprogram_declarative_part
    begin
        subprogram_statement_part
    end [ subprogram_kind ] [ designator ] ;
```

```
subprogram_declarative_part ::=
    { subprogram_declarative_item }

subprogram_declarative_item ::=
      subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | variable_declaration
    | file_declaration
    | alias_declaration
    | attribute_declaration
    | attribute_specification
    | use_clause
    | group_template_declaration
    | group_declaration

subprogram_statement_part ::=
    { sequential_statement }

subprogram_kind ::=  procedure | function
```

The declaration of a subprogram is optional. In the absence of such a declaration, the subprogram specification of the subprogram body acts as the declaration. For each subprogram declaration, there must be a corresponding body. If both a declaration and a body are given, the subprogram specification of the body must conform (see 2.7) to the subprogram specification of the declaration. Furthermore, both the declaration and the body must occur immediately within the same declarative region (see 10.1).

If a subprogram kind appears at the end of a subprogram body, it must repeat the reserved word given in the subprogram specification. If a designator appears at the end of a subprogram body, it must repeat the designator of the subprogram.

It is an error if a variable declaration in a subprogram declarative part declares a shared variable. (See 4.3.1.3 and 8.1.4.)

A *foreign subprogram* is one that is decorated with the attribute 'FOREIGN, defined in package STANDARD (see 14.2). The STRING value of the attribute may specify implementation-dependent information about the foreign subprogram. Foreign subprograms may have non-VHDL implementations. An implementation may place restrictions on the allowable modes, classes, and types of the formal parameters to a foreign subprogram; such restrictions may include restrictions on the number and allowable order of the parameters.

Excepting foreign subprograms, the algorithm performed by a subprogram is defined by the sequence of statements that appears in the subprogram statement part. For a foreign subprogram, the algorithm performed is implementation defined.

The execution of a subprogram body is invoked by a subprogram call. For this execution, after establishing the association between the formal and actual parameters, the sequence of statements of the body is executed if the subprogram is not a foreign subprogram; otherwise, an implementation-defined action occurs. Upon completion of the body or implementation-dependent action, if exclusive access to an object of a protected type was granted during elaboration of the declaration of the subprogram (see 12.5), the exclusive access is rescinded. Then, return is made to the caller (and any necessary copying back of formal to actual parameters occurs).

A process or a subprogram is said to be a *parent* of a given subprogram S if that process or subprogram contains a procedure call or function call for S or for a parent of S.

An *explicit signal* is a signal other than an implicit signal GUARD or other than one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION. The *explicit ancestor* of an

implicit signal is found as follows. The implicit signal GUARD has no explicit ancestor. An explicit ancestor of an implicit signal defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION is the signal found by recursively examining the prefix of the attribute. If the prefix denotes an explicit signal, a slice, or a member (see ~~Section~~ Clause[7] 3) of an explicit signal, then that is the explicit ancestor of the implicit signal. Otherwise, if the prefix is one of the implicit signals defined by the predefined attributes 'DELAYED, 'STABLE, 'QUIET, or 'TRANSACTION, this rule is recursively applied. If the prefix is an implicit signal GUARD, then the signal has no explicit ancestor.

If a pure function subprogram is a parent of a given procedure and if that procedure contains a reference to an explicitly declared signal or variable object, or a slice or subelement (or slice thereof) of an explicit signal, then that object must be declared within the declarative region formed by the function (see 10.1) or within the declarative region formed by the procedure; this rule also holds for the explicit ancestor, if any, of an implicit signal and also for the implicit signal GUARD. If a pure function is the parent of a given procedure, then that procedure must not contain a reference to an explicitly declared file object (see 4.3.1.4) or to a shared variable (see 4.3.1.3).

Similarly, if a pure function subprogram contains a reference to an explicitly declared signal or variable object, or a slice or subelement (or slice thereof) of an explicit signal, then that object must be declared within the declarative region formed by the function; this rule also holds for the explicit ancestor, if any, of an implicit signal and also for the implicit signal GUARD. A pure function must not contain a reference to an explicitly declared file object.

A pure function must not be the parent of an impure function.

The rules of the preceding ~~four~~ three[8] paragraphs apply to all pure function subprograms. For pure functions that are not foreign subprograms, violations of any of these rules are errors. However, since implementations cannot in general check that such rules hold for pure function subprograms that are foreign subprograms, a description calling pure foreign function subprograms not adhering to these rules is erroneous.

Example:

-- The declaration of a foreign function subprogram:

```
package P is
        function F return INTEGER;
        attribute FOREIGN of F: function is "implementation-dependent information";
end package P;
```

NOTES

1—It follows from the visibility rules that a subprogram declaration must be given if a call of the subprogram occurs textually before the subprogram body, and that such a declaration must occur before the call itself.

2—The preceding rules concerning pure function subprograms, together with the fact that function parameters ~~may only~~ must[9] be of mode **in**, imply that a pure function has no effect other than the computation of the returned value. Thus, a pure function invoked explicitly as part of the elaboration of a declaration, or one invoked implicitly as part of the simulation cycle, is guaranteed to have no effect on other objects in the description.

3—VHDL does not define the parameter-passing mechanisms for foreign subprograms.

4—The declarative parts and statement parts of subprograms decorated with the 'FOREIGN attribute are subject to special elaboration rules. See 12.3 and 12.4.

---

7. To conform to IEEE rules.
8. Typo noted by Steve Bailey.
9. IR1000.4.7.

5—A pure function subprogram ~~may~~ must[10] not reference a shared variable. This prohibition exists because a shared variable ~~may not~~ cannot[11] be declared in a subprogram declarative part and a pure function ~~may not~~ cannot[12] reference any variable declared outside of its declarative region.

6—A subprogram containing a wait statement must not have an ancestor that is a subprogram declared within either a protected type declaration or a protected type body.

2.3 Subprogram overloading

Two formal parameter lists are said to have the same *parameter type profile* if and only if they have the same number of parameters, and if at each parameter position the corresponding parameters have the same base type. Two subprograms are said to have the same *parameter and result type profile* if and only if both have the same parameter type profile, and if either both are functions with the same result base type or neither of the two is a function.

A given subprogram designator can be used ~~in several subprogram specifications~~ to designate multiple subprograms[13]. The subprogram designator is then said to be overloaded; the designated subprograms are also said to be overloaded and to overload each other. If two subprograms overload each other, one of them can hide the other only if both subprograms have the same parameter and result type profile.

A call to an overloaded subprogram is ambiguous (and therefore is an error) if the name of the subprogram, the number of parameter associations, the types and order of the actual parameters, the names of the formal parameters (if named associations are used), and the result type (for functions) are not sufficient to identify exactly one (overloaded) subprogram ~~specification~~[14].

Similarly, a reference to an overloaded resolution function name in a subtype indication is ambiguous (and is therefore an error) if the name of the function, the number of formal parameters, the result type, and the relationships between the result type and the types of the formal parameters (as defined in 2.4) are not sufficient to identify exactly one (overloaded) subprogram specification.

Examples:

--Declarations of overloaded subprograms:

```
procedure Dump(F: inout Text; Value: Integer);
procedure Dump(F: inout Text; Value: String);

procedure Check (Setup: Time; signal D: Data; signal C: Clock);
procedure Check (Hold: Time; signal C: Clock; signal D: Data);
```

--Calls to overloaded subprograms:

```
Dump (Sys_Output, 12) ;
Dump (Sys_Error, "Actual output does not match expected output") ;

Check (Setup=>10 ns, D=>DataBus, C=>Clk1) ;
Check (Hold=>5 ns, D=>DataBus, C=>Clk2);
Check (15 ns, DataBus, Clk) ;
    -- Ambiguous if the base type of DataBus is the same type as the base type of Clk.
```

10. IR1000.4.7.
11. IR1000.4.7.
12. IR1000.4.7.
13. IR1000.4.3.
14. IR1000.2.7.

NOTES

1—The notion of parameter and result type profile does not include parameter names, parameter classes, parameter modes, parameter subtypes, or default expressions or their presence or absence.

2—Ambiguities may (but need not) arise when actual parameters of the call of an overloaded subprogram are themselves overloaded function calls, literals, or aggregates. Ambiguities may also (but need not) arise when several overloaded subprograms belonging to different packages are visible. These ambiguities can usually be solved in two ways: qualified expressions can be used for some or all actual parameters and for the result, if any; or the name of the subprogram can be expressed more explicitly as an expanded name (see 6.3).

2.3.1 Operator overloading

The declaration of a function whose designator is an operator symbol is used to overload an operator. The sequence of characters of the operator symbol must be one of the operators in the operator classes defined in 7.2.

The subprogram specification of a unary operator must have a single parameter, unless the subprogram specification is a method (see 3.5.1) of a protected type. In this latter case, the subprogram specification must have no parameters[15]. The subprogram specification of a binary operator must have two parameters~~;~~, unless the subprogram specification is a method of a protected type, in which case, the subprogram specification must have a single parameter. If the subprogram specification of a binary operator has two parameters,[16] for each use of this operator, the first parameter is associated with the left operand, and the second parameter is associated with the right operand.

For each of the operators "+" and "–", overloading is allowed both as a unary operator and as a binary operator.

NOTES

1—Overloading of the equality operator does not affect the selection of choices in a case statement in a selected signal assignment statement; nor does it ~~have an affect on~~ affect[17] the propagation of signal values.

2—A user-defined operator that has the same designator as a short-circuit operator (that is, that overloads the short-circuit operator) is not invoked in a short-circuit manner. Specifically, calls to the user-defined operator always evaluate both arguments prior to the execution of the function.

3—Functions that overload operator symbols may also be called using function call notation rather than operator notation. This statement is also true of the predefined operators themselves.

Examples:

```
type MVL is ('0', '1', 'Z', 'X') ;

function "and" (Left, Right: MVL) return MVL ;
function "or" (Left, Right: MVL) return MVL ;
function "not" (Value: MVL) return MVL ;

signal Q,R,S: MVL ;

Q <= 'X' or '1';
R <= "or" ('0','Z');
S <= (Q and R) or not S;
```

15. LCS 26.
16. LCS 26.
17. IR1000.1.4, as modified by Ashenden.

2.3.2 Signatures

A signature distinguishes between overloaded subprograms and overloaded enumeration literals based on their parameter and result type profiles. A signature can be used in an attribute name, entity designator, or alias declaration.

signature ::= [[type_mark { , type_mark }] [**return** type_mark]]

(Note that the initial and terminal brackets are part of the syntax of signatures and do not indicate that the entire right-hand side of the production is optional.) A signature is said to *match* the parameter and result type profile of a given subprogram if and only if all of the following conditions hold:

— The number of type marks prior to the reserved word **return**, if any, matches the number of formal parameters of the subprogram

— At each parameter position, the base type denoted by the type mark of the signature is the same as the base type of the corresponding formal parameter of the subprogram

— If the reserved word **return** is present, the subprogram is a function and the base type of the type mark following the reserved word in the signature is the same as the base type of the return type of the function, or the reserved word **return** is absent and the subprogram is a procedure

Similarly, a signature is said to match the parameter and result type profile of a given enumeration literal if the signature matches the parameter and result type profile of the subprogram equivalent to the enumeration literal defined in 3.1.1.

Example:

attribute BuiltIn **of** "or" [MVL, MVL **return** MVL]: **function is** TRUE;
-- Because of the presence of the signature, this attribute specification
-- decorates only the "or" function defined in the previous ~~section~~ subclause[18].

attribute Mapping **of** JMP [**return** OpCode] : **literal is** “001”;

2.4 Resolution functions

A resolution function is a function that defines how the values of multiple sources of a given signal are to be resolved into a single value for that signal. Resolution functions are associated with signals that require resolution by including the name of the resolution function in the declaration of the signal or in the declaration of the subtype of the signal. A signal with an associated resolution function is called a resolved signal (see 4.3.1.2).

A resolution function must be a pure function (see 2.1); moreover, it must have a single input parameter of class **constant** that is a one-dimensional, unconstrained array whose element type is that of the resolved signal. The type of the return value of the function must also be that of the signal. Errors occur at the place of the subtype indication containing the name of the resolution function if any of these checks fail (see 4.2).

The resolution function associated with a resolved signal determines the *resolved value* of the signal as a function of the collection of inputs from its multiple sources. If a resolved signal is of a composite type, and if subelements of that type also have associated resolution functions, such resolution functions have no effect on the process of determining the resolved value of the signal. It is an error if a resolved signal has more connected sources than the number of elements in the index type of the unconstrained array type used to define the parameter of the corresponding resolution function.

Resolution functions are implicitly invoked during each simulation cycle in which corresponding resolved signals are active (see 12.6.1). Each time a resolution function is invoked, it is passed an array value, each element of

18. To conform to IEEE rules.

which is determined by a corresponding source of the resolved signal, but excluding those sources that are drivers whose values are determined by null transactions (see 8.4.1). Such drivers are said to be *off*. For certain invocations (specifically, those involving the resolution of sources of a signal declared with the signal kind **bus**), a resolution function may thus be invoked with an input parameter that is a null array; this occurs when all sources of the bus are drivers, and they are all off. In such a case, the resolution function returns a value representing the value of the bus when no source is driving it.

Example:

```
function WIRED_OR (Inputs: BIT_VECTOR) return BIT is
    constant FloatValue: BIT := '0';
begin
    if Inputs'Length = 0 then
        -- This is a bus whose drivers are all off.
        return FloatValue;
    else
        for I in Inputs'Range loop
            if Inputs(I) = '1' then
                return '1';
            end if;
        end loop;
        return '0';
    end if;
end function WIRED_OR;
```

2.5 Package declarations

A package declaration defines the interface to a package. The scope of a declaration within a package can be extended to other design units.

```
package_declaration ::=
    package identifier is
        package_declarative_part
    end [ package ] [ package_simple_name ] ;

package_declarative_part ::=
    { package_declarative_item }

package_declarative_item ::=
      subprogram_declaration
    | type_declaration
    | subtype_declaration
    | constant_declaration
    | signal_declaration
    | shared_variable_declaration
    | file_declaration
    | alias_declaration
    | component_declaration
    | attribute_declaration
    | attribute_specification
    | disconnection_specification
    | use_clause
    | group_template_declaration
    | group_declaration
```

If a simple name appears at the end of the package declaration, it must repeat the identifier of the package declaration.

If a package declarative item is a type declaration that is a full type declaration whose type definition is a protected_type definition, then that protected type definition must not be a protected type body.

Items declared immediately within a package declaration become visible by selection within a given design unit wherever the name of that package is visible in the given unit. Such items may also be made directly visible by an appropriate use clause (see 10.4).

NOTE

—Not all packages will have a package body. In particular, a package body is unnecessary if no subprograms, deferred constants, or protected type definitions are declared in the package declaration.

Examples:

--A package declaration that needs no package body:

```
package TimeConstants is
    constant tPLH :Time := 10 ns;
    constant tPHL :Time := 12 ns;
    constant tPLZ :Time := 7 ns;
    constant tPZL :Time := 8 ns;
    constant tPHZ :Time := 8 ns;
    constant tPZH :Time := 9 ns;
end TimeConstants ;
```

--A package declaration that needs a package body:

```
package TriState is
    type Tri is ('0', '1', 'Z', 'E');
    function BitVal (Value: Tri) return Bit ;
    function TriVal (Value: Bit) return Tri;
    type TriVector is array (Natural range <>) of Tri ;
    function Resolve (Sources: TriVector) return Tri ;
end package TriState ;
```

2.6 Package bodies

A package body defines the bodies of subprograms and the values of deferred constants declared in the interface to the package.

```
package_body ::=
    package body package_simple_name is
        package_body_declarative_part
    end [ package body ] [ package_simple_name ] ;

package_body_declarative_part ::=
    { package_body_declarative_item }
```

```
package_body_declarative_item ::=
        subprogram_declaration
      | subprogram_body
      | type_declaration
      | subtype_declaration
      | constant_declaration
      | shared_variable_declaration
      | file_declaration
      | alias_declaration
      | use_clause
      | group_template_declaration
      | group_declaration
```

The simple name at the start of a package body must repeat the package identifier. If a simple name appears at the end of the package body, it must be the same as the identifier in the package declaration.

In addition to subprogram body and constant declarative items, a package body may contain certain other declarative items to facilitate the definition of the bodies of subprograms declared in the interface. Items declared in the body of a package cannot be made visible outside of the package body.

If a given package declaration contains a deferred constant declaration (see 4.3.1.1), then a constant declaration with the same identifier must appear as a declarative item in the corresponding package body. This object declaration is called the *full* declaration of the deferred constant. The subtype indication given in the full declaration must conform to that given in the deferred constant declaration.

Within a package declaration that contains the declaration of a deferred constant, and within the body of that package (before the end of the corresponding full declaration), the use of a name that denotes the deferred constant is only allowed in the default expression for a local generic, local port, or formal parameter. The result of evaluating an expression that references a deferred constant before the elaboration of the corresponding full declaration is not defined by the language.

Example:

```
package body TriState is

    function BitVal (Value: Tri) return Bit is
        constant Bits : Bit_Vector := "0100";
    begin
        return Bits(Tri'Pos(Value));
    end;

    function TriVal (Value: Bit) return Tri is
    begin
        return Tri'Val(Bit'Pos(Value));
    end;
```

```
    function Resolve (Sources: TriVector) return Tri is
        variable V: Tri := 'Z';
    begin
        for i in Sources'Range loop
            if Sources(i) /= 'Z' then
                if V = 'Z' then
                    V := Sources(i);
                else
                    return 'E';
                end if;
            end if;
        end loop;
        return V;
    end;

end package body TriState ;
```

2.7 Conformance rules

Whenever the language rules either require or allow the specification of a given subprogram to be provided in more than one place, the following variations are allowed at each place:

— A numeric literal can be replaced by a different numeric literal if and only if both have the same value.

— A simple name can be replaced by an expanded name in which this simple name is the selector if and only if at both places the meaning of the simple name is given by the same declaration.

Two subprogram specifications are said to *conform* if, apart from comments and the above allowed variations, both specifications are formed by the same sequence of lexical elements and if corresponding lexical elements are given the same meaning by the visibility rules.

Conformance is likewise defined for subtype indications in deferred constant declarations.

NOTES

1—A simple name can be replaced by an expanded name even if the simple name is itself the prefix of a selected name. For example, Q.R can be replaced by P.Q.R if Q is declared immediately within P.

2—The subprogram specification of an impure function is never conformant to a subprogram specification of a pure function.

3—The following specifications do not conform since they are not formed by the same sequence of lexical elements:

```
procedure P (X,Y : INTEGER)
procedure P (X: INTEGER; Y : INTEGER)
procedure P (X,Y : in INTEGER)
```