Clause 8

Sequential statements

The various forms of sequential statements are described in this ~~section~~ clause[1]. Sequential statements are used to define algorithms for the execution of a subprogram or process; they execute in the order in which they appear.

```
sequence_of_statements ::=
    { sequential_statement }

sequential_statement ::=
      wait_statement
    | assertion_statement
    | report_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | procedure_call_statement
    | if_statement
    | case_statement
    | loop_statement
    | next_statement
    | exit_statement
    | return_statement
    | null_statement
```

All sequential statements may be labeled. Such labels are implicitly declared at the beginning of the declarative part of the innermost enclosing process statement or subprogram body.

8.1 Wait statement

The wait statement causes the suspension of a process statement or a procedure.

```
wait_statement ::=
    [ label : ] wait [ sensitivity_clause ] [ condition_clause ] [ timeout_clause ] ;

sensitivity_clause ::= on sensitivity_list

sensitivity_list ::= signal_name { , signal_name }

condition_clause ::= until condition

condition ::= boolean_expression

timeout_clause ::= for time_expression
```

1. To conform to IEEE rules.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

The sensitivity clause defines the *sensitivity set* of the wait statement, which is the set of signals to which the wait statement is sensitive. Each signal name in the sensitivity list identifies a given signal as a member of the sensitivity set. Each signal name in the sensitivity list must be a static signal name, and each name must denote a signal for which reading is permitted. If no sensitivity clause appears, the sensitivity set is constructed according to the following (recursive) rule:

The sensitivity set is initially empty. For each primary in the condition of the condition clause, if the primary is

- A simple name that denotes a signal, add the longest static prefix of the name to the sensitivity set
- A selected name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set
- ~~An expanded name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set~~[2]
- An indexed name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set and apply this rule to all expressions in the indexed name
- A slice name whose prefix denotes a signal, add the longest static prefix of the name to the sensitivity set and apply this rule to any expressions appearing in the discrete range of the slice name
- An attribute name, if the designator denotes a signal attribute, add the longest static prefix of the name of the implicit signal denoted by the attribute name to the sensitivity set; otherwise, apply this rule to the prefix of the attribute name
- An aggregate, apply this rule to every expression appearing after the choices and the =>, if any, in every element association
- A function call, apply this rule to every actual designator in every parameter association
- An actual designator of **open** in a parameter association, do not add to the sensitivity set
- A qualified expression, apply this rule to the expression or aggregate qualified by the type mark, as appropriate
- A type conversion, apply this rule to the expression type converted by the type mark
- A parenthesized expression, apply this rule to the expression enclosed within the parentheses
- Otherwise, do not add to the sensitivity set

This rule is also used to construct the sensitivity sets of the wait statements in the equivalent process statements for concurrent procedure call statements (9.3), concurrent assertion statements (9.4), and concurrent signal assignment statements (9.5).

If a signal name that denotes a signal of a composite type appears in a sensitivity list, the effect is as if the name of each scalar subelement of that signal appears in the list.

The condition clause specifies a condition that must be met for the process to continue execution. If no condition clause appears, the condition clause **until** TRUE is assumed.

The timeout clause specifies the maximum amount of time the process will remain suspended at this wait statement. If no timeout clause appears, the timeout clause **for** (STD.STANDARD.TIME'HIGH – STD.STANDARD.NOW) is assumed. It is an error if the time expression in the timeout clause evaluates to a negative value.

2. Simplification noted by Paul Graham. An expanded name can never have a prefix denoting a signal.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

The execution of a wait statement causes the time expression to be evaluated to determine the *timeout interval*. It also causes the execution of the corresponding process statement to be suspended, where the corresponding process statement is the one that either contains the wait statement or is the parent (see 2.2) of the procedure that contains the wait statement. The suspended process will resume, at the latest, immediately after the timeout interval has expired.

The suspended process ~~may~~ can[3] also resume as a result of an event occurring on any signal in the sensitivity set of the wait statement. If such an event occurs, the condition in the condition clause is evaluated. If the value of the condition is TRUE, the process will resume. If the value of the condition is FALSE, the process will re-suspend. Such re-suspension does not involve the recalculation of the timeout interval.

It is an error if a wait statement appears in a function subprogram or in a procedure that has a parent that is a function subprogram. Furthermore, it is an error if a wait statement appears in an explicit process statement that includes a sensitivity list or in a procedure that has a parent that is such a process statement. Finally, it is an error if a wait statement appears within any subprogram whose body is declared within a protected type body, or within any subprogram that has an ancestor whose body is declared within a protected type body.

Example:

```
type Arr is array (1 to 5) of BOOLEAN;
function F (P: BOOLEAN) return BOOLEAN;
signal S: Arr;
signal l, r: INTEGER range 1 to 5;

-- The following two wait statements have the same meaning:

wait until F(S(3)) and (S(l) or S(r));
wait on S(3), S, l, r until F(S(3)) and (S(l) or S(r));
```

NOTES

1—The wait statement **wait until** Clk = '1'; has semantics identical to

```
loop
    wait on Clk;
    exit when Clk = '1';
end loop;
```

because of the rules for the construction of the default sensitivity clause. These same rules imply that **wait until** True; has semantics identical to **wait**;.

2—The conditions that cause a wait statement to resume execution of its enclosing process may no longer hold at the time the process resumes execution if the enclosing process is a postponed process.

3—The rule for the construction of the default sensitivity set implies that if a function call appears in a condition clause and the called function is an impure function, then any signals that are accessed by the function but that are not passed through the association list of the call are not added to the default sensitivity set for the condition by virtue of the appearance of the function call in the condition.

8.2 Assertion statement

An assertion statement checks that a specified condition is true and reports an error if it is not.

assertion_statement ::= [label :] assertion ;

3. IR1000.4.7.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
assertion ::=
    assert condition
        [ report expression ]
        [ severity expression ]
```

If the **report** clause is present, it must include an expression of predefined type STRING that specifies a message to be reported. If the **severity** clause is present, it must specify an expression of predefined type SEVERITY_LEVEL that specifies the severity level of the assertion.

The **report** clause specifies a message string to be included in error messages generated by the assertion. In the absence of a **report** clause for a given assertion, the string "Assertion violation." is the default value for the message string. The **severity** clause specifies a severity level associated with the assertion. In the absence of a **severity** clause for a given assertion, the default value of the severity level is ERROR.

Evaluation of an assertion statement consists of evaluation of the Boolean expression specifying the condition. If the expression results in the value FALSE, then an *assertion violation* is said to occur. When an assertion violation occurs, the **report** and **severity** clause expressions of the corresponding assertion, if present, are evaluated. The specified message string and severity level (or the corresponding default values, if not specified) are then used to construct an error message.

The error message consists of at least

a) An indication that this message is from an assertion

b) The value of the severity level

c) The value of the message string

d) The name of the design unit (see 11.1) containing the assertion

8.3 Report statement

A report statement displays a message.

```
report_statement ::=
    [ label : ]
        report expression
            [ severity expression ] ;
```

The **report** statement expression must be of the predefined type STRING. The string value of this expression is included in the message generated by the report statement. If the **severity** clause is present, it must specify an expression of predefined type SEVERITY_LEVEL. The severity clause specifies a severity level associated with the report. In the absence of a **severity** clause for a given report, the default value of the severity level is NOTE.

The evaluation of a report statement consists of the evaluation of the report expression and severity clause expression, if present. The specified message string and severity level (or corresponding default, if the severity level is not specified) are then used to construct a report message.

The report message consists of at least

a) An indication that this message is from a report statement

b) The value of the severity level

c) The value of the message string

d) The name of the design unit containing the report statement

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Examples:

```
report "Entering process P";                          -- A report statement
                                                      -- with default severity NOTE.

report "Setup or Hold violation; outputs driven to 'X'"   -- Another report statement;
    severity WARNING;                                 -- severity is specified.
```

8.4 Signal assignment statement

A signal assignment statement modifies the projected output waveforms contained in the drivers of one or more signals (see 12.6.1).

```
signal_assignment_statement ::=
    [ label : ] target <= [ delay_mechanism ] waveform ;

delay_mechanism ::=
        transport
    | [ reject time_expression ] inertial

target ::=
        name
    | aggregate

waveform ::=
        waveform_element { , waveform_element }
    | unaffected
```

If the target of the signal assignment statement is a name, then the name must denote a signal, and the base type of the value component of each transaction produced by a waveform element on the right-hand side must be the same as the base type of the signal denoted by that name. This form of signal assignment assigns right-hand side values to the drivers associated with a single (scalar or composite) signal.

If the target of the signal assignment statement is in the form of an aggregate, then the type of the aggregate must be determinable from the context, excluding the aggregate itself but including the fact that the type of the aggregate must be a composite type. The base type of the value component of each transaction produced by a waveform element on the right-hand side must be the same as the base type of the aggregate. Furthermore, the expression in each element association of the aggregate must be a locally static name that denotes a signal. This form of signal assignment assigns slices or subelements of the right-hand side values to the drivers associated with the signal named as the corresponding slice or subelement of the aggregate.

If the target of a signal assignment statement is in the form of an aggregate, and if the expression in an element association of that aggregate is a signal name that denotes a given signal, then the given signal and each subelement thereof (if any) are said to be *identified* by that element association as targets of the assignment statement. It is an error if a given signal or any subelement thereof is identified as a target by more than one element association in such an aggregate. Furthermore, it is an error if an element association in such an aggregate contains an **others** choice or a choice that is a discrete range.

The right-hand side of a signal assignment may optionally specify a delay mechanism. A delay mechanism consisting of the reserved word **transport** specifies that the delay associated with the first waveform element is to be construed as *transport* delay. Transport delay is characteristic of hardware devices (such as transmission lines) that exhibit nearly infinite frequency response: any pulse is transmitted, no matter how short its duration. If no delay mechanism is present, or if a delay mechanism including the reserved word **inertial** is present, the delay is construed to be *inertial* delay. Inertial delay is characteristic of switching circuits: a pulse whose duration is shorter than the switching time of the circuit will not be transmitted, or in the case that a pulse rejection limit is specified, a pulse whose duration is shorter than that limit will not be transmitted.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Every inertially delayed signal assignment has a *pulse rejection limit*. If the delay mechanism specifies inertial delay, and if the reserved word **reject** followed by a time expression is present, then the time expression specifies the pulse rejection limit. In all other cases, the pulse rejection limit is specified by the time expression associated with the first waveform element.

It is an error if the pulse rejection limit for any inertially delayed signal assignment statement is either negative or greater than the time expression associated with the first waveform element.

It is an error if the reserved word **unaffected** appears as a waveform in a (sequential) signal assignment statement.

NOTES

1— The reserved word **unaffected** ~~may~~ must[4] only appear as a waveform in concurrent signal assignment statements. See 9.5.1.

2— For a signal assignment whose target is a name, the type of the target ~~may~~ must[5] not be a protected type~~, nor may the target have a subelement whose type~~ ; moreover, it is an error if the type of any subelement of the target[6] is a protected type.

3— For a signal assignment whose target is in the form of an aggregate, ~~no~~ it is an error if any[7] element of the target ~~may be~~ is[8] of a protected type~~, nor may any element of the target have a subelement whose type~~ ; moreover, it is an error if the type of any element of the target has a subelement that[9] is a protected type.

Examples:

```
-- Assignments using inertial delay:

    -- The following three assignments are equivalent to each other:

        Output_pin <= Input_pin after 10 ns;
        Output_pin <= inertial Input_pin after 10 ns;
        Output_pin <= reject 10 ns inertial Input_pin after 10 ns;

    -- Assignments with a pulse rejection limit less than the time expression:

        Output_pin <= reject 5 ns inertial Input_pin after 10 ns;
        Output_pin <= reject 5 ns inertial Input_pin after 10 ns, not Input_pin after 20 ns;

-- Assignments using transport delay:

        Output_pin <= transport Input_pin after 10 ns;
        Output_pin <= transport Input_pin after 10 ns, not Input_pin after 20 ns;

    -- Their equivalent assignments:

        Output_pin <= reject 0 ns inertial Input_pin after 10 ns;
        Output_pin <= reject 0 ns inertial Input_pin after 10 ns, not Input_pin after ~~10~~ 20[10] ns;
```

4. IR1000.4.7.
5. IR1000.4.7.
6. IR1000.4.7.
7. IR1000.4.7.
8. IR1000.4.7.
9. IR1000.4.7.
10. Typo. correction, as noted by D. Borrione during P1076a balloting.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

NOTE

—If a right-hand side value expression is either a numeric literal or an attribute that yields a result of type *universal_integer* or *universal_real*, then an implicit type conversion is performed.

8.4.1 Updating a projected output waveform

The effect of execution of a signal assignment statement is defined in terms of its effect upon the projected output waveforms (see 12.6.1) representing the current and future values of drivers of signals.

waveform_element ::=
*value*_expression [**after** *time*_expression]
| **null** [**after** *time*_expression]

The future behavior of the driver(s) for a given target is defined by transactions produced by the evaluation of waveform elements in the waveform of a signal assignment statement. The first form of waveform element is used to specify that the driver is to assign a particular value to the target at the specified time. The second form of waveform element is used to specify that the driver of the signal is to be turned off, so that it (at least temporarily) stops contributing to the value of the target. This form of waveform element is called a *null waveform element*. It is an error if the target of a signal assignment statement containing a null waveform element is not a guarded signal or an aggregate of guarded signals.

The base type of the time expression in each waveform element must be the predefined physical type TIME as defined in package STANDARD. If the **after** clause of a waveform element is not present, then an implicit "**after** 0 ns" is assumed. It is an error if the time expression in a waveform element evaluates to a negative value.

Evaluation of a waveform element produces a single transaction. The time component of the transaction is determined by the current time added to the value of the time expression in the waveform element. For the first form of waveform element, the value component of the transaction is determined by the value expression in the waveform element. For the second form of waveform element, the value component is not defined by the language, but it is defined to be of the type of the target. A transaction produced by the evaluation of the second form of waveform element is called a *null transaction*.

For the execution of a signal assignment statement whose target is of a scalar type, the waveform on its right-hand side is first evaluated. Evaluation of a waveform consists of the evaluation of each waveform element in the waveform. Thus, the evaluation of a waveform results in a sequence of transactions, where each transaction corresponds to one waveform element in the waveform. These transactions are called *new* transactions. It is an error if the sequence of new transactions is not in ascending order with respect to time.

The sequence of transactions is then used to update the projected output waveform representing the current and future values of the driver associated with the signal assignment statement. Updating a projected output waveform consists of the deletion of zero or more previously computed transactions (called *old* transactions) from the projected output waveform and the addition of the new transactions, as follows:

— All old transactions that are projected to occur at or after the time at which the earliest new transaction is projected to occur are deleted from the projected output waveform.

— The new transactions are then appended to the projected output waveform in the order of their projected occurrence.

If the initial delay is inertial delay according to the definitions of 8.4, the projected output waveform is further modified as follows:

a) All of the new transactions are marked;

b) An old transaction is marked if the time at which it is projected to occur is less than the time at which the first new transaction is projected to occur minus the pulse rejection limit;

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

c) For each remaining unmarked, old transaction, the old transaction is marked if it immediately precedes a marked transaction and its value component is the same as that of the marked transaction;

d) The transaction that determines the current value of the driver is marked;

e) All unmarked transactions (all of which are old transactions) are deleted from the projected output waveform.

For the purposes of marking transactions, any two successive null transactions in a projected output waveform are considered to have the same value component.

The execution of a signal assignment statement whose target is of a composite type proceeds in a similar fashion, except that the evaluation of the waveform results in one sequence of transactions for each scalar subelement of the type of the target. Each such sequence consists of transactions whose value portions are determined by the values of the same scalar subelement of the value expressions in the waveform, and whose time portion is determined by the time expression corresponding to that value expression. Each such sequence is then used to update the projected output waveform of the driver of the matching subelement of the target. This applies both to a target that is the name of a signal of a composite type and to a target that is in the form of an aggregate.

If a given procedure is declared by a declarative item that is not contained within a process statement, and if a signal assignment statement appears in that procedure, then the target of the assignment statement must be a formal parameter of the given procedure or of a parent of that procedure, or an aggregate of such formal parameters. Similarly, if a given procedure is declared by a declarative item that is not contained within a process statement, and if a signal is associated with an **inout** or **out** mode signal parameter in a subprogram call within that procedure, then the signal so associated must be a formal parameter of the given procedure or of a parent of that procedure.

NOTES

1—These rules guarantee that the driver affected by a signal assignment statement is always statically determinable if the signal assignment appears within a given process (including the case in which it appears within a procedure that is declared within the given process). In this case, the affected driver is the one defined by the process; otherwise, the signal assignment must appear within a procedure, and the affected driver is the one passed to the procedure along with a signal parameter of that procedure.

2—Overloading the operator "=" has no effect on the updating of a projected output waveform.

3—Consider a signal assignment statement of the form

T <= **reject** t_r **inertial** e_1 **after** t_1 { , e_i **after** t_i } ;

The following relations hold:

$$0\ \text{ns} \le t_r \le t_1$$

and

$$0\ \text{ns} \le t_i < t_{i+1}$$

Note that, if $t_r = 0$ ns, then the waveform editing is identical to that for transport-delayed assignment, and if $t_r = t_1$, the waveform is identical to that for the statement

T <= e_1 **after** t_1 { , e_i **after** t_i } ;

4—Consider the following signal assignment in some process:

S <= **reject** 15 ns **inertial** 12 **after** 20 ns, 18 **after** 41 ns;

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

where S is a signal of some integer type. Assume that at the time this signal assignment is executed, the driver of S in the process has the following contents (the first entry is the current driving value):

1	2	2	12	5	8
NOW	+3 ns	+12 ns	+13 ns	+20 ns	+42 ns

(The times given are relative to the current time.) The updating of the projected output waveform proceeds as follows:

a. The driver is truncated at 20 ns. The driver now contains the following pending transactions:

1	2	2	12
NOW	+3 ns	+12 ns	+13 ns

b. The new waveforms are added to the driver. The driver now contains the following pending transactions:

1	2	2	12	12	18
NOW	+3 ns	+12 ns	+13 ns	+20 ns	+41 ns

c. All new transactions are marked, as well as those old transactions that occur at less than the time of the first new waveform (20 ns) less the rejection limit (15 ns). The driver now contains the following pending transactions (marked transactions are emboldened):

1	**2**	2	12	**12**	**18**
NOW	**+3 ns**	+12 ns	+13 ns	**+20 ns**	**+41 ns**

d. Each remaining unmarked transaction is marked if it immediately precedes a marked transaction and has the same value as the marked transaction. The driver now contains the following pending transactions:

1	**2**	2	**12**	**12**	**18**
NOW	**+3 ns**	+12 ns	**+13 ns**	**+20 ns**	**+41 ns**

e. The transaction that determines the current value of the driver is marked, and all unmarked transactions are then deleted. The final driver contents are then as follows, after clearing the markings:

1	2	12	12	18
NOW	+3 ns	+13 ns	+20 ns	+41 ns

5—No subtype check is performed on the value component of a new transaction when it is added to a driver. Instead, a subtype check that the value component of a transaction belongs to the subtype of the signal driven by the driver is made when the driver takes on that value. See 12.6.1.

8.5 Variable assignment statement

A variable assignment statement replaces the current value of a variable with a new value specified by an expression. The named variable and the right-hand side expression must be of the same type.

variable_assignment_statement ::=
    [label :] target := expression ;

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

If the target of the variable assignment statement is a name, then the name must denote a variable, and the base type of the expression on the right-hand side must be the same as the base type of the variable denoted by that name. This form of variable assignment assigns the right-hand side value to a single (scalar or composite) variable.

If the target of the variable assignment statement is in the form of an aggregate, then the type of the aggregate must be determinable from the context, excluding the aggregate itself but including the fact that the type of the aggregate must be a composite type. The base type of the expression on the right-hand side must be the same as the base type of the aggregate. Furthermore, the expression in each element association of the aggregate must be a locally static name that denotes a variable. This form of variable assignment assigns each subelement or slice of the right-hand side value to the variable named as the corresponding subelement or slice of the aggregate.

If the target of a variable assignment statement is in the form of an aggregate, and if the locally static name in an element association of that aggregate denotes a given variable or denotes another variable of which the given variable is a subelement or slice, then the element association is said to *identify* the given variable as a target of the assignment statement. It is an error if a given variable is identified as a target by more than one element association in such an aggregate.

For the execution of a variable assignment whose target is a variable name, the variable name and the expression are first evaluated. A check is then made that the value of the expression belongs to the subtype of the variable, except in the case of a variable that is an array (in which case the assignment involves a subtype conversion). Finally, the value of the expression becomes the new value of the variable. A design is erroneous if it depends on the order of evaluation of the target and source expressions of an assignment statement.

The execution of a variable assignment whose target is in the form of an aggregate proceeds in a similar fashion, except that each of the names in the aggregate is evaluated, and a subtype check is performed for each subelement or slice of the right-hand side value that corresponds to one of the names in the aggregate. The value of the subelement or slice of the right-hand side value then becomes the new value of the variable denoted by the corresponding name.

An error occurs if the aforementioned subtype checks fail.

The determination of the type of the target of a variable assignment statement may require determination of the type of the expression if the target is a name that can be interpreted as the name of a variable designated by the access value returned by a function call, and similarly, as an element or slice of such a variable.

NOTES

1— If the right-hand side is either a numeric literal or an attribute that yields a result of type *universal integer* or *universal real*, then an implicit type conversion is performed.

2— For a variable assignment whose target is a name, the type of the target ~~may~~ must[11] not be a protected type~~, nor may the target have a subelement whose type~~ ; moreover, it is an error if the type of any subelement of the target[12] is a protected type.

3— For a variable assignment whose target is in the form of an aggregate, ~~no~~ it is an error if any[13] element of the target ~~may be~~ is[14] of a protected type~~, nor may any element of the target have a subelement whose type~~ ; moreover, it is an error if the type of any element of the target has a subelement that[15] is a protected type.

11. IR1000.4.7.
12. IR1000.4.7.
13. IR1000.4.7.
14. IR1000.4.7.
15. IR1000.4.7.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

8.5.1 Array variable assignments

If the target of an assignment statement is a name denoting an array variable (including a slice), the value assigned to the target is implicitly converted to the subtype of the array variable; the result of this subtype conversion becomes the new value of the array variable.

This means that the new value of each element of the array variable is specified by the matching element (see 7.2.2) in the corresponding array value obtained by evaluation of the expression. The subtype conversion checks that for each element of the array variable there is a matching element in the array value, and vice versa. An error occurs if this check fails.

NOTE

—The implicit subtype conversion described for assignment to an array variable is performed only for the value of the right-hand side expression as a whole; it is not performed for subelements or slices that are array values.

8.6 Procedure call statement

A procedure call invokes the execution of a procedure body.

procedure_call_statement ::= [label :] procedure_call ;

procedure_call ::= *procedure*_name [(actual_parameter_part)]

The procedure name specifies the procedure body to be invoked. The actual parameter part, if present, specifies the association of actual parameters with formal parameters of the procedure.

For each formal parameter of a procedure, a procedure call must specify exactly one corresponding actual parameter. This actual parameter is specified either explicitly, by an association element (other than the actual **open**) in the association list or, in the absence of such an association element, by a default expression (see 4.3.2).

Execution of a procedure call includes evaluation of the actual parameter expressions specified in the call and evaluation of the default expressions associated with formal parameters of the procedure that do not have actual parameters associated with them. In both cases, the resulting value must belong to the subtype of the associated formal parameter. (If the formal parameter is of an unconstrained array type, then the formal parameter takes on the subtype of the actual parameter.) The procedure body is executed using the actual parameter values and default expression values as the values of the corresponding formal parameters.

8.7 If statement

An if statement selects for execution one or none of the enclosed sequences of statements, depending on the value of one or more corresponding conditions.

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

If a label appears at the end of an if statement, it must repeat the if label.

For the execution of an if statement, the condition specified after **if**, and any conditions specified after **elsif**, are evaluated in succession (treating a final **else** as **elsif** TRUE **then**) until one evaluates to TRUE or all conditions

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

are evaluated and yield FALSE. If one condition evaluates to TRUE, then the corresponding sequence of statements is executed; otherwise, none of the sequences of statements is executed.

8.8 Case statement

A case statement selects for execution one of a number of alternative sequences of statements; the chosen alternative is defined by the value of an expression.

```
case_statement ::=
    [ case_label : ]
        case expression is
            case_statement_alternative
            { case_statement_alternative }
        end case [ case_label ] ;

case_statement_alternative ::=
    when choices =>
        sequence_of_statements
```

The expression must be of a discrete type, or of a one-dimensional array type whose element base type is a character type. This type must be determinable independently of the context in which the expression occurs, but using the fact that the expression must be of a discrete type or a one-dimensional character array type. Each choice in a case statement alternative must be of the same type as the expression; the list of choices specifies for which values of the expression the alternative is chosen.

If the expression is the name of an object whose subtype is locally static, whether a scalar type or an array type, then each value of the subtype must be represented once and only once in the set of choices of the case statement, and no other value is allowed; this rule is likewise applied if the expression is a qualified expression or type conversion whose type mark denotes a locally static subtype, or if the expression is a call to a function whose return type mark denotes a locally static subtype.

If the expression is of a one-dimensional character array type, then the expression must be one of the following:

— The name of an object whose subtype is locally static

— An indexed name whose prefix is one of the members of this list and whose indexing expressions are locally static expressions

— A slice name whose prefix is one of the members of this list and whose discrete range is a locally static discrete range

— A function call whose return type mark denotes a locally static subtype

— A qualified expression or type conversion whose type mark denotes a locally static subtype

In such a case, each choice appearing in any of the case statement alternatives must be a locally static expression whose value is of the same length as that of the case expression. It is an error if the element subtype of the one-dimensional character array type is not a locally static subtype.

For other forms of expression, each value of the (base) type of the expression must be represented once and only once in the set of choices, and no other value is allowed.

The simple expression and discrete ranges given as choices in a case statement must be locally static. A choice defined by a discrete range stands for all values in the corresponding range. The choice **others** is only allowed for the last alternative and as its only choice; it stands for all values (possibly none) not given in the choices of previous alternatives. An element simple name (see 7.3.2) is not allowed as a choice of a case statement alternative.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

If a label appears at the end of a case statement, it must repeat the case label.

The execution of a case statement consists of the evaluation of the expression followed by the execution of the chosen sequence of statements. A sequence of statements in a given case statement alternative is the chosen sequence of statements if and only if the expression “E = V” evaluates to True, where “E” is the expression, “V” is the value of one of the choices of the given case statement alternative (if a choice is a discrete range, then this latter condition is fulfilled when V is an element of the discrete range), and the operator "=" in the expression is the predefined "=" operator on the base type of E.[16]

NOTES

1—The execution of a case statement chooses one and only one alternative, since the choices are exhaustive and mutually exclusive. A qualified expression whose type mark denotes a locally static subtype can often be used as the expression of a case statement to limit the number of choices that need be explicitly specified.

2—An **others** choice is required in a case statement if the type of the expression is the type *universal_integer* (for example, if the expression is an integer literal), since this is the only way to cover all values of the type *universal_integer*.

3—Overloading the operator “=” has no effect on the semantics of case statement execution.

8.9 Loop statement

A loop statement includes a sequence of statements that is to be executed repeatedly, zero or more times.

```
loop_statement ::=
    [ loop_label : ]
        [ iteration_scheme ] loop
            sequence_of_statements
        end loop [ loop_label ] ;

iteration_scheme ::=
        while condition
    | for loop_parameter_specification

parameter_specification ::=
    identifier in discrete_range
```

If a label appears at the end of a loop statement, it must repeat the label at the beginning of the loop statement.

Execution of a loop statement is complete when the loop is left as a consequence of the completion of the iteration scheme (see below), if any, or the execution of a next statement, an exit statement, or a return statement.

A loop statement without an iteration scheme specifies repeated execution of the sequence of statements.

For a loop statement with a **while** iteration scheme, the condition is evaluated before each execution of the sequence of statements; if the value of the condition is TRUE, the sequence of statements is executed; if FALSE, the iteration scheme is said to be *complete* and the execution of the loop statement is complete.

For a loop statement with a **for** iteration scheme, the loop parameter specification is the declaration of the *loop parameter* with the given identifier. The loop parameter is an object whose type is the base type of the discrete range. Within the sequence of statements, the loop parameter is a constant. Hence, a loop parameter is not allowed as the target of an assignment statement. Similarly, the loop parameter must not be given as an actual corresponding to a formal of mode **out** or **inout** in an association list.

For the execution of a loop with a **for** iteration scheme, the discrete range is first evaluated. If the discrete range is a null range, the iteration scheme is said to be *complete* and the execution of the loop statement is therefore

16. LCS 14.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

complete; otherwise, the sequence of statements is executed once for each value of the discrete range (subject to the loop not being left as a consequence of the execution of a next statement, an exit statement, or a return statement), after which the iteration scheme is said to be *complete*. Prior to each such iteration, the corresponding value of the discrete range is assigned to the loop parameter. These values are assigned in left-to-right order.

NOTE

—A loop may be left as the result of the execution of a next statement if the loop is nested inside of an outer loop and the next statement has a loop label that denotes the outer loop.

8.10 Next statement

A next statement is used to complete the execution of one of the iterations of an enclosing loop statement (called "loop" in the following text). The completion is conditional if the statement includes a condition.

next_statement ::=
    [label :] **next** [*loop*_label] [**when** condition] ;

A next statement with a loop label is only allowed within the labeled loop and applies to that loop; a next statement without a loop label is only allowed within a loop and applies only to the innermost enclosing loop (whether labeled or not).

For the execution of a next statement, the condition, if present, is first evaluated. The current iteration of the loop is terminated if the value of the condition is TRUE or if there is no condition.

8.11 Exit statement

An exit statement is used to complete the execution of an enclosing loop statement (called "loop" in the following text). The completion is conditional if the statement includes a condition.

exit_statement ::=
    [label :] **exit** [*loop*_label] [**when** condition] ;

An exit statement with a loop label is only allowed within the labeled loop and applies to that loop; an exit statement without a loop label is only allowed within a loop and applies only to the innermost enclosing loop (whether labeled or not).

For the execution of an exit statement, the condition, if present, is first evaluated. Exit from the loop then takes place if the value of the condition is TRUE or if there is no condition.

8.12 Return statement

A return statement is used to complete the execution of the innermost enclosing function or procedure body.

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

A return statement is only allowed within the body of a function or procedure, and it applies to the innermost enclosing function or procedure.

A return statement appearing in a procedure body must not have an expression. A return statement appearing in a function body must have an expression.

The value of the expression defines the result returned by the function. The type of this expression must be the base type of the type mark given after the reserved word **return** in the specification of the function. It is an error if execution of a function completes by any means other than the execution of a return statement.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

For the execution of a return statement, the expression (if any) is first evaluated and a check is made that the value belongs to the result subtype. The execution of the return statement is thereby completed if the check succeeds; so also is the execution of the enclosing subprogram. An error occurs at the place of the return statement if the check fails.

NOTES

1—If the expression is either a numeric literal, or an attribute that yields a result of type *universal_integer* or *universal_real*, then an implicit conversion of the result is performed.

2—If the return type mark of a function denotes a constrained array subtype, then no implicit subtype conversions are performed on the values of the expressions of the return statements within the subprogram body of that function. Thus, for each index position of each value, the bounds of the discrete range must be the same as the discrete range of the return subtype, and the directions must be the same.

8.13 Null statement

A null statement performs no action.

null_statement ::=
[label :] **null** ;

The execution of the null statement has no effect other than to pass on to the next statement.

NOTE

—The null statement can be used to specify explicitly that no action is to be performed when certain conditions are true, although it is never mandatory for this (or any other) purpose. This is particularly useful in conjunction with the case statement, in which all possible values of the case expression must be covered by choices: for certain choices, it may be that no action is required.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

Copyright © 2000, IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.