Clause 10

Scope and visibility

The rules defining the scope of declarations and the rules defining which identifiers are visible at various points in the text of the description are presented in this ~~section~~ clause[1]. The formulation of these rules uses the notion of a declarative region.

10.1 Declarative region~~A~~ With two exceptions, a[2] declarative region is a portion of the text of the description. A single declarative region is formed by the text of each of the following:

a) An entity declaration~~, together with a corresponding architecture body~~[3].

b) An architecture body.[4]

~~b~~ c) A configuration declaration.

~~c~~ d) A subprogram declaration, together with the corresponding subprogram body.

~~d~~ e) A package declaration, together with the corresponding body (if any).

~~e~~ f) A record type declaration.

~~f~~ g) A component declaration.

~~g~~ h) A block statement.

~~h~~ i) A process statement.

~~i~~ j) A loop statement.

~~j~~ k) A block configuration.

~~k~~ l) A component configuration.

~~l~~ m) A generate statement.

~~m~~ n) A protected type declaration, together with the corresponding body.

1. To conform to IEEE rules.
2. LCS 3.
3. LCS 3.
4. LCS 3.

In each of these cases, the declarative region is said to be *associated* with the corresponding declaration or statement. A declaration is said to occur *immediately within* a declarative region if this region is the innermost region that encloses the declaration, not counting the declarative region (if any) associated with the declaration itself.

Certain declarative regions include disjoint parts. Each declarative region is nevertheless considered as a (logically) continuous portion of the description text. Hence, if any rule defines a portion of text as the text that *extends* from some specific point of a declarative region to the end of this region, then this portion is the corresponding subset of the declarative region (thus, it does not include intermediate declarative items between the interface declaration and a corresponding body declaration).

In addition to the above declarative regions, there is a *root declarative region*, not associated with a portion of the text of the description, but encompassing any given primary unit. At the beginning of the analysis of a given primary unit, there are no declarations whose scopes (see 10.2) are within the root declarative region. Moreover, the root declarative region associated with any given secondary unit is the root declarative region of the corresponding primary unit.

There is also a *library declarative region* associated with each design library (see 11.2). Each library declarative region has within its scope declarations corresponding to each primary unit contained within the associated design library.

The declarative region associated with an architecture body is considered to occur immediately within the declarative region associated with the entity declaration corresponding to the given architecture body.

NOTE

—The fact that an architecture body has an associated root declarative region does not mean that the declarative region associated with the architecture is directly within the associated root declarative region. Instead, the declarative region associated with the corresponding entity declaration surrounds the declarative region associated with the architecture.[5]

10.2 Scope of declarations

For each form of declaration, the language rules define a certain portion of the description text called the *scope of the declaration*. The scope of a declaration is also called the scope of any named entity declared by the declaration. Furthermore, if the declaration associates some notation (either an identifier, a character literal, or an operator symbol) with the named entity, this portion of the text is also called the scope of this notation. Within the scope of a named entity, and only there, there are places where it is legal to use the associated notation in order to refer to the named entity. These places are defined by the rules of visibility and overloading.

The scope of a declaration ~~that occurs immediately within a declarative region~~[6] extends from the beginning of the declaration to the end of the immediately enclosing[7] declarative region; this part of the scope of a declaration is called the *immediate scope*. Furthermore, for any of the declarations in the following list, the scope of the declaration extends beyond the immediate scope:

a) A declaration that occurs immediately within a package declaration

b) An element declaration in a record type declaration

c) A formal parameter declaration in a subprogram declaration

d) A local generic declaration in a component declaration

e) A local port declaration in a component declaration

5. LCS 3.
6. LCS 3.
7. LCS 3.

f) A formal generic declaration in an entity declaration

g) A formal port declaration in an entity declaration

h) A declaration that occurs immediately within a protected type declaration

In the absence of a separate subprogram declaration, the subprogram specification given in the subprogram body acts as the declaration, and rule c) applies also in such a case. In each of these cases, the given declaration occurs immediately within some enclosing declaration, and the scope of the given declaration extends to the end of the scope of the enclosing declaration.

In addition to the above rules, the scope of any declaration that includes the end of the declarative part of a given block (whether it be an external block defined by a design entity or an internal block defined by a block statement) extends into a configuration declaration that configures the given block.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if the scope of a given declaration includes the end of the declarative part of that block, then the scope of the given declaration extends from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block. Furthermore, the scope of a use clause is similarly extended. Finally, the scope of a library unit contained within a design library is extended along with the scope of the logical library name corresponding to that design library.

NOTE

—These scope rules apply to all forms of declaration. In particular, they apply also to implicit declarations and to named design units[8].

10.3 Visibility

The meaning of the occurrence of an identifier at a given place in the text is defined by the visibility rules and also, in the case of overloaded declarations, by the overloading rules. The identifiers considered in this ~~section~~ subclause[9] include any identifier other than a reserved word or an[10] attribute designator that denotes a predefined attribute. The places considered in this ~~section~~ subclause[11] are those where a lexical element (such as an identifier) occurs. The overloaded declarations considered in this ~~section~~ subclause[12] are those for subprograms and enumeration literals.

For each identifier and at each place in the text, the visibility rules determine a set of declarations (with this identifier) that define the possible meanings of an occurrence of the identifier. A declaration is said to be *visible* at a given place in the text when, according to the visibility rules, the declaration defines a possible meaning of this occurrence. Two cases ~~may~~[13] arise in determining the meaning of such a declaration:

— The visibility rules determine *at most one* possible meaning. In such a case, the visibility rules are sufficient to determine the declaration defining the meaning of the occurrence of the identifier, or in the absence of such a declaration, to determine that the occurrence is not legal at the given point.

— The visibility rules determine *more than one* possible meaning. In such a case, the occurrence of the identifier is legal at this point if and only if *exactly one* visible declaration is acceptable for the overloading rules in the given context.

8. LCS 3.
9. To conform to IEEE rules.
10. IR1106.1.1.
11. To conform to IEEE rules.
12. To conform to IEEE rules.
13. IR1000.4.7.

A declaration is ~~only visible~~ visible only[14] within a certain part of its scope; this part starts at the end of the declaration except in the declaration of a design unit or a protected type declaration, in which case it starts immediately after the reserved word **is** occuring after the identifier of the design unit or protected type declaration. This rule applies to both explicit and implicit declarations.

Visibility is either by selection or direct. A declaration is visible *by selection* at places that are defined as follows:

a) For a primary unit contained in a library: at the place of the suffix in a selected name whose prefix denotes the library.

b) For an entity name in a configuration declaration whose entity name is a simple name: at the place of the simple name, and the context is that of the library "Work".[15]

~~b~~ c) For an architecture body associated with a given entity declaration: at the place of the block specification in a block configuration for an external block whose interface is defined by that entity declaration.

~~c~~ d) For an architecture body associated with a given entity declaration: at the place of an architecture identifier (between the parentheses) in the first form of an entity aspect in a binding indication.

~~d~~ e) For a declaration given in a package declaration: at the place of the suffix in a selected name whose prefix denotes the package.

~~e~~ f) For an element declaration of a given record type declaration: at the place of the suffix in a selected name whose prefix is appropriate for the type; also at the place of a choice (before the compound delimiter =>) in a named element association of an aggregate of the type.

~~f~~ g) For a user-defined attribute: at the place of the attribute designator (after the delimiter ') in an attribute name whose prefix denotes a named entity with which that attribute has been associated.

~~g~~ h) For a formal parameter declaration of a given subprogram declaration: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named parameter association element of a corresponding subprogram call.

~~h~~ i) For a local generic declaration of a given component declaration: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named generic association element of a corresponding component instantiation statement; similarly, at the place of the actual designator in an actual part (after the compound delimiter =>, if any) of a generic association element of a corresponding binding indication.

~~i~~ j) For a local port declaration of a given component declaration: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named port association element of a corresponding component instantiation statement; similarly, at the place of the actual designator in an actual part (after the compound delimiter =>, if any) of a port association element of a corresponding binding indication.

~~j~~ k) For a formal generic declaration of a given entity declaration: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named generic association element of a corresponding binding indication; similarly, at the place of the formal designator in a formal part (before the compound delimiter =>) of a generic association element of a corresponding component instantiation statement when the instantiated unit is a design entity or a configuration declaration.

~~k~~ l) For a formal port declaration of a given entity declaration: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named port association element of a corresponding binding specification; similarly, at the place of the formal designator in a formal part (before the

14. IR1106.1.2.
15. LCS 3.

compound delimiter =>) of a port association element of a corresponding component instantiation statement when the instantiated unit is a design entity or a configuration declaration.

~~l~~ m) For a formal generic declaration or a formal port declaration of a given block statement: at the place of the formal designator in a formal part (before the compound delimiter =>) of a named association element of a corresponding generic or port map aspect.

~~m~~ n) For a subprogram declared immediately within a given protected type declaration: at the place of the suffix in a selected name whose prefix denotes an object of the protected type.

Finally, within the declarative region associated with a construct other than a record type declaration or a protected type any declaration that occurs immediately within the region and that also occurs textually within the construct is visible by selection at the place of the suffix of an expanded name whose prefix denotes the construct.

Where it is not visible by selection, a visible declaration is said to be *directly visible*. A declaration is said to be directly visible within a certain part of its immediate scope; this part extends to the end of the immediate scope of the declaration but excludes places where the declaration is hidden as explained in the following paragraphs. In addition, a declaration occurring immediately within the visible part of a package can be made directly visible by means of a use clause according to the rules described in 10.4.

A declaration is said to be *hidden* within (part of) an inner declarative region if the inner region contains a homograph of this declaration; the outer declaration is then hidden within the immediate scope of the inner homograph. Each of two declarations is said to be a *homograph* of the other if both declarations have the same identifier, operator symbol, or character literal, and if overloading is allowed for at most one of the two. If overloading is allowed for both declarations, then each of the two is a homograph of the other if they have the same identifier, operator symbol, or character literal, as well as the same parameter and result type profile (see 3.1.1).

Within the specification of a subprogram, every declaration with the same designator as the subprogram is hidden. Where hidden in this manner, a declaration is visible neither by selection nor directly.

Two declarations that occur immediately within the same declarative region must not be homographs, unless exactly one of them is the implicit declaration of a predefined operation. In such cases, a predefined operation is always hidden by the other homograph. Where hidden in this manner, an implicit declaration is hidden within the entire scope of the other declaration (regardless of which declaration occurs first); the implicit declaration is visible neither by selection nor directly.

Whenever a declaration with a certain identifier is visible from a given point, the identifier and the named entity (if any) are also said to be visible from that point. Direct visibility and visibility by selection are likewise defined for character literals and operator symbols. An operator is directly visible if and only if the corresponding operator declaration is directly visible.

In addition to the aforementioned rules, any declaration that is visible by selection at the end of the declarative part of a given (external or internal) block is visible by selection in a configuration declaration that configures the given block.

In addition, any declaration that is directly visible at the end of the declarative part of a given block is directly visible in a block configuration that configures the given block. This rule holds unless a use clause that makes a homograph of the declaration potentially visible (see 10.4) appears in the corresponding configuration declaration, and if the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the declaration will be directly visible within the corresponding configuration items, except at those places that fall within the scope of the additional use clause. At such places, neither name will be directly visible.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if a given declaration is visible by selection at the end of the declarative part of that block, then the given declaration is visible by selection from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block.

If a component configuration appears as a configuration item immediately within a block configuration that configures a given block, and if a given declaration is directly visible at the end of the declarative part of that block, then the given declaration is visible by selection from the beginning to the end of the declarative region associated with the given component configuration. A similar rule applies to a block configuration that appears as a configuration item immediately within another block configuration, provided that the contained block configuration configures an internal block. Furthermore, the visibility of declarations made directly visible by a use clause within a block is similarly extended. Finally, the visibility of a logical library name corresponding to a design library directly visible at the end of a block is similarly extended. The rules of this paragraph hold unless a use clause that makes a homograph of the declaration potentially visible appears in the corresponding block configuration, and if the scope of that use clause encompasses all or part of those configuration items. If such a use clause appears, then the declaration will be directly visible within the corresponding configuration items, except at those places that fall within the scope of the additional use clause. At such places, neither name will be directly visible.

NOTES

1—The same identifier, character literal, or operator symbol may occur in different declarations and may thus be associated with different named entities, even if the scopes of these declarations overlap. Overlap of the scopes of declarations with the same identifier, character literal, or operator symbol can result from overloading of subprograms and of enumeration literals. Such overlaps can also occur for named entities declared in the visible parts of packages and for formal generics and ports, record elements, and formal parameters, where there is overlap of the scopes of the enclosing package declarations, entity ~~interfaces~~ declarations[16], record type declarations, or subprogram declarations. Finally, overlapping scopes can result from nesting.

2—The rules defining immediate scope, hiding, and visibility imply that a reference to an identifier, character literal, or operator symbol within its own declaration is illegal (except for design units). The identifier, character literal, or operator symbol hides outer homographs within its immediate scope—that is, from the start of the declaration. On the other hand, the identifier, character literal, or operator symbol is visible only after the end of the declaration (again, except for design units). For this reason, all but the last of the following declarations are illegal:

```
constant K: INTEGER := K*K;          -- Illegal
constant T: T;                       -- Illegal
procedure P (X: P);                  -- Illegal
function Q (X: REAL := Q) return Q;  -- Illegal
procedure R (R: REAL);               -- Legal (although perhaps confusing)
```

Example:

```
L1: block
        signal A,B: Bit ;
    begin
    L2: block
            signal B: Bit ;                  -- An inner homograph of B.
        begin
            A <= B after 5 ns;               -- Means L1.A <= L2.B
            B <= L1.B after 10 ns;           -- Means L2.B <= L1.B
        end block ;
        B <= A after 15 ns;                  -- Means L1.B <= L1.A
    end block ;
```

10.4 Use clauses

A use clause achieves direct visibility of declarations that are visible by selection.

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

16. Terminological correction.

Each selected name in a use clause identifies one or more declarations that will potentially become directly visible. If the suffix of the selected name is a simple name, character literal, or operator symbol, then the selected name identifies only the declaration(s) of that simple name, character literal, or operator symbol contained within the package or library denoted by the prefix of the selected name. If the suffix is the reserved word **all**, then the selected name identifies all declarations that are contained within the package or library denoted by the prefix of the selected name.

For each use clause, there is a certain region of text called the *scope* of the use clause. This region starts immediately after the use clause. If a use clause is a declarative item of some declarative region, the scope of the clause extends to the end of the given[17] declarative region. If a use clause occurs within the context clause of a design unit, the scope of the use clause extends to the end of the root[18] declarative region associated with the given[19] design unit. The scope of a use clause may additionally extend into a configuration declaration (see 10.2).

In order to determine which declarations are made directly visible at a given place by use clauses, consider the set of declarations identified by all use clauses whose scopes enclose this place. Any declaration in this set is a potentially visible declaration. A potentially visible declaration is actually made directly visible except in the following two cases:

a) A potentially visible declaration is not made directly visible if the place considered is within the immediate scope of a homograph of the declaration.

b) Potentially visible declarations that have the same designator are not made directly visible unless each of them is either an enumeration literal specification or the declaration of a subprogram (either by a subprogram declaration or by an implicit declaration).

NOTES

1—These rules guarantee that a declaration that is made directly visible by a use clause cannot hide an otherwise directly visible declaration.

2—If a named entity X declared in package P is made potentially visible within a package Q (e.g., by the inclusion of the clause "**use** P.X;" in the context clause of package Q), and the context clause for design unit R includes the clause "**use** Q.**all**;", this does not imply that X will be potentially visible in R. Only those named entities that are actually declared in package Q will be potentially visible in design unit R (in the absence of any other use clauses).

10.5 The context of overload resolution

Overloading is defined for names, subprograms, and enumeration literals.

For overloaded entities, overload resolution determines the actual meaning that an occurrence of an identifier or a character literal has whenever the visibility rules have determined that more than one meaning is acceptable at the place of this occurrence; overload resolution likewise determines the actual meaning of an occurrence of an operator or basic operation (see the introduction to ~~Section~~ Clause[20] 3).

At such a place, all visible declarations are considered. The occurrence is only legal if there is exactly one interpretation of each constituent of the innermost complete context; a *complete context* is either a declaration, a specification, or a statement.

When considering possible interpretations of a complete context, the only rules considered are the syntax rules, the scope and visibility rules, and the rules of the form described below.

a) Any rule that requires a name or expression to have a certain type or to have the same type as another name or expression.

17. Clarification.
18. LCS 3.
19. Clarification.
20. To conform to IEEE rules.

b) Any rule that requires the type of a name or expression to be a type of a certain class; similarly, any rule that requires a certain type to be a discrete, integer, floating point, physical, universal, ~~character, or Boolean~~ or character[21] type.

c) Any rule that requires a prefix to be appropriate for a certain type.

d) The rules that require the type of an aggregate or string literal to be determinable solely from the enclosing complete context. Similarly, the rules that require the type of the prefix of an attribute, the type of the expression of a case statement, or the type of the operand of a type conversion to be determinable independently of the context.

e) The rules given for the resolution of overloaded subprogram calls; for the implicit conversions of universal expressions; for the interpretation of discrete ranges with bounds having a universal type; and for the interpretation of an expanded name whose prefix denotes a subprogram.

f) The rules given for the requirements on the return type, the number of formal parameters, and the types of the formal parameters of the subprogram denoted by the resolution function name (see 2.4).

NOTES

1—If there is only one possible interpretation of an occurrence of an identifier, character literal, operator symbol, or string, that occurrence denotes the corresponding named entity. However, this condition does not mean that the occurrence is necessarily legal since other requirements exist that are not considered for overload resolution: for example, the fact that the expression is static, the parameter modes, conformance rules, the use of named association in an indexed name, the use of **open** in an indexed name, the use of a slice as an actual to a function call, and so forth.

2—A loop parameter specification is a declaration, and hence a complete context.

3—Rules that require certain constructs to have the same parameter and result type profile fall under category a) above. The same holds for rules that require conformance of two constructs, since conformance requires that corresponding names be given the same meaning by the visibility and overloading rules.

21. IR1000.4.2.