interface_nonansi_header ::=
{ attribute_instance } **interface** [lifetime] interface_identifier
{ package_import_declaration } [parameter_port_list] list_of_ports ;

interface_ansi_header ::=
{attribute_instance } **interface** [lifetime] interface_identifier
{ package_import_declaration }[1] [parameter_port_list] [list_of_port_declarations] ;

program_nonansi_header ::=
{ attribute_instance } **program** [lifetime] program_identifier
{ package_import_declaration } [parameter_port_list] list_of_ports ;

program_ansi_header ::=
{attribute_instance } **program** [lifetime] program_identifier
{ package_import_declaration }[1] [parameter_port_list] [list_of_port_declarations] ;

1) A package_import_declaration in a module_ansi_header, interface_ansi_header, or program_ansi_header shall be followed by a parameter_port_list or list_of_port_declarations, or both.

Syntax 26-3—Package import in header syntax (excerpt from Annex A)

Package items that are imported as part of a module, interface, or program header are visible throughout the module, interface, or program, including in parameter and port declarations.

For example:

```
package A;
   typedef struct {
      bit [ 7:0] opcode;
      bit [23:0] addr;
   } instruction_t;
endpackage: A

package B;
   typedef enum bit {FALSE, TRUE} boolean_t;
endpackage: B

module M import A::instruction_t, B::*;
         #(WIDTH = 32)
          (input  [WIDTH-1:0]    data,
           input  instruction_t  a,
           output [WIDTH-1:0]    result,
           output boolean_t      OK
          );
  ...
endmodule: M
```

26.5 Search order rules

Table 26-1 describes the search order rules for the declarations imported from a package. For the purposes of the discussion that follows, consider the following package declarations:

```
package p;
   typedef enum { FALSE, TRUE } BOOL;
   const BOOL c = FALSE;
endpackage
```

```
package q;
   const int c = 0;
endpackage
```

Table 26-1—Scoping rules for package importation

Example	Description	In a scope containing a local declaration of c	In a scope not containing a local declaration of c	In a scope containing an explicit import of c (import q::c)	In a scope containing a wildcard import of c (import q::*)
`u = p::c;` `y = p::TRUE;`	A qualified package identifier is visible in any scope (without the need for an import clause).	OK Direct reference to `c` refers to the locally declared `c`. `p::c` refers to the `c` in package `p`.	OK Direct reference to `c` is illegal because it is undefined. `p::c` refers to the `c` in package `p`.	OK Direct reference to `c` refers to the `c` imported from `q`. `p::c` refers to the `c` in package `p`.	OK Direct reference to `c` refers to the `c` imported from `q`. `p::c` refers to the `c` in package `p`.

`import` `p::*;`
 	
`. . .`
 	
`y = FALSE;`
	All declarations inside package `p` become potentially directly visible in the importing scope:
	– `c`
	– `BOOL`
	– `FALSE`
	– `TRUE`
		OK
		
		Direct reference to `c` refers to the locally declared `c`.
		
		Direct reference to other identifiers (e.g., `FALSE`) refers to those implicitly imported from package `p`.
			OK
			
			Direct reference to `c` refers to the `c` imported from package `p`.
				OK
				
				Direct reference to `c` refers to the `c` imported from package `q`.
					OK / ERROR
					
					`c` is undefined in the importing scope. Thus, a direct reference to `c` is illegal and results in an error.
					
					The import clause is otherwise allowed.

`import` `p::c;`
 	
`. . .`
 	
`if( ! c ) ...`
	The imported identifier `c` becomes directly visible in the importing scope.	ERROR
		
		It is illegal to import an identifier defined in the importing scope.
			OK
			
			Direct reference to `c` refers to the `c` imported from package `p`.
				ERROR
				
				It is illegal to import the same identifier from different packages.
					OK / ERROR
					
					The import of `p::c` makes any prior reference to `c` illegal.
					
					Otherwise, direct reference to `c` refers to the `c` imported from package `p`.

When using a wildcard import, a reference to an undefined identifier that is declared within the package causes that identifier to be imported into the scope of the import statement. However, an error results if the same identifier is later declared or explicitly imported in the same scope. This is shown in the following example:

```
module m;
   import q::*;
```

```
   wire   a = c;  // This statement forces the import of q::c;
   import p::c;   // The conflict with q::c and p::c creates an error.
endmodule
```

26.6 Exporting imported names from packages

By default, declarations imported into a package are not visible by way of subsequent imports of that package. Package export declarations allow a package to specify that imported declarations are to be made visible in subsequent imports. A package export may precede a corresponding package import.

The syntax for package exports is shown in Syntax 26-4.

---

package_export_declaration ::= *// from A.2.1.3*
  export *::* ;
| **export** package_import_item { , package_import_item } ;

Syntax 26-4—Package export syntax (excerpt from Annex A)

An export of the form `package_name::*` exports all declarations that were actually imported from `package_name` within the context of the exporting package. All names from `package_name`, whether imported directly or through wildcard imports, are made available. Symbols that are candidates for import but not actually imported are not made available. The special wildcard export form, `export *::*;` , exports all imported declarations from all packages from which imports occur.

An export of the form `package_name::name` makes the given declaration available. It shall be an error if the given declaration is not a candidate for import or if the declaration is not actually imported in the package. The declaration being exported shall be imported from the same `package_name` used in the export. If the declaration is an unreferenced candidate for import, the export shall be considered to be a reference and shall import the declaration into the package following the same rules as for a direct import of the name.

An import of a declaration made visible through an export is equivalent to an import of the original declaration. Thus direct or wildcard import of a declaration by way of multiple exported paths does not cause conflicts.

It is valid to specify multiple exports that export the same actual declaration.

Examples:

```
package p1;
   int x, y;
endpackage

package p2;
   import p1::x;
   export p1::*;     // exports p1::x as the name "x";
                     // p1::x and p2::x are the same declaration
endpackage

package p3;
   import p1::*;
   import p2::*;
   export p2::*;
```

```
   int q = x;

   // p1::x and q are made available from p3. Although p1::y
   // is a candidate for import, it is not actually imported
   // since it is not referenced. Since p1::y is not imported,
   // it is not made available by the export.
endpackage

package p4;
   import p1::*;
   export p1::*;
   int y = x;          // y is available as a direct declaration;
                       // p1::x is made available by the export
endpackage

package p5;
   import p4::*;
   import p1::*;
   export p1::x;
   export p4::x;       // p4::x refers to the same declaration
                       // as p1::x so this is legal.
endpackage

package p6;
   import p1::*;
   export p1::x;
   int x;              // Error. export p1::x is considered to
                       // be a reference to "x" so a subsequent
                       // declaration of x is illegal.
endpackage

package p7;
   int y;
endpackage

package p8;
   export *::*;        // Exports both p7::y and p1::x.
   import p7::y;
   import p1::x;
endpackage

module top;
   import p2::*;
   import p4::*;
   int y = x;          // x is p1::x
endmodule
```

26.7 The std built-in package

SystemVerilog provides a built-in package that can contain system types (e.g., classes), variables, tasks, and functions. Users cannot insert additional declarations into the built-in package.

The contents of the standard built-in package are defined in Annex G.