

- A **Program** is an organized collection of program units. There must be exactly one main program, and in addition there may be modules, external subprograms, and block data units. Elements described by means other than Fortran may be included.
 - A **Module** provides a means of packaging related data and procedures, and hiding information not needed outside the module. There are several intrinsic modules.
 - The **Data Environment** consists of the data objects upon which operations will be performed to create desired results or values. These objects may have declared and dynamic types; they may have type parameters, and they may possess attributes such as dimensionality. They need not exist for the whole execution of the program. Allocatable objects and pointer targets may be created when needed and released when no longer needed.
 - **Program Execution** begins with the first executable construct in the main program and continues with successive constructs unless there is a change in the flow of control. When a procedure is invoked, its execution begins with its first executable construct. On normal return, execution continues where it left off. Execution may occur simultaneously with input/output processes.
 - The **Definition Status** of a variable indicates whether or not the variable has a value; the value may change during execution. Most variables are initially undefined and become defined when they acquire a value. The status also may become undefined during execution. Pointers have both an association status and a definition status. Allocatable objects have both an allocation status and a definition status.
 - **Scope** and **Association** determine where and by what names various entities are known and accessible in a program. These concepts form the information backbone of the language.
-

This chapter introduces the basic concepts and fundamental terms needed to understand Fortran. Some terms are defined implicitly by the syntax rules. Others, such as “associated” or “present” are ordinary English words, but they have a specific Fortran meaning.

One of the major concepts involves the organization of a program. A program consists of program units; program units consist of Fortran statements. Some statements are executable; some are not. In general, the nonexecutable statements define the data environment, and the executable statements specify the actions taken. This chapter presents the high-level syntax rules for a Fortran program. It also describes the order in which constructs and statements may appear in a program and concludes with an example of a short, but complete, Fortran program.

While there is some discussion of language features here to help explain various terms and concepts, Chapters 3–16 contain the complete description of all language features.

2.1 Program Organization

A collection of program units constitutes an executable program. A Fortran program must have one main program and may have any number of the other program units. Program units may serve as hosts for smaller scoping units. Information may be hidden within part of a program or communicated to other parts of a program by various means. The programmer may control the parts of a program in which information is accessible.

With the introduction of C interoperability in Fortran 2003, it is possible to include, with much greater ease and portability, external procedures and other entities defined by a means other than Fortran. A processor has one or more companion processors. A **companion processor** is a processor-dependent mechanism by which global data and procedures may be referenced or defined. It may be the Fortran processor itself, or it may be another Fortran processor. If a procedure is defined by means of a companion processor that is not the Fortran processor itself, the standard refers to the C function that defines the procedure. Although the procedure need not be defined by means of the C programming language, the interoperability mechanisms are designed to mesh well with C.

2.1.1 Program Units

A Fortran program unit is one of the following:

- main program
- module
- external subprogram
- block data

A Fortran program may consist of only a main program, although usually there are also modules and/or external subprograms which may be subroutine or function subprograms. These program units contain constructs and statements that define the data environment and the steps necessary to perform calculations. Each program unit has an END statement to terminate the program unit. Each has a special initial statement as well, but the initial statement for a main program is optional. For example, a program might contain a main program, a module, and a subroutine:

```

program task
  . . .
  call calc (z)
  . . .
end program task

```

```

module info
  . . .
end module info

subroutine calc(x)
  use info
  . . .
end subroutine calc

```

An ideal Fortran program would consist of a main program and several modules; that is, there would be no external subprograms. This is the best model for packaging and encapsulation (2.2.5). Subroutine and function subprograms are a fundamental part of the language. They may be module, internal, or external subprograms.

The interface of a procedure supplies information about the name and type (if a function) of the procedure, as well as information about its arguments. A program is more robust if the interfaces of procedures are known when the procedures are invoked. This is inherently the case for internal procedures, module procedures, and all of the intrinsic procedures. In addition, the interfaces of procedures defined in other languages must be described to the Fortran system as C function interfaces (15.6).

The main program could be defined in a language other than Fortran, but it is usually the language of the main program that determines the program's primary nature. For example, a Fortran main program with some elements specified in another language is still a Fortran program; whereas, if the main program is specified in C but there is access to Fortran elements, the program is generally considered to be a C program. Interlanguage communication is described in 15.

Because all except the most trivial of programs will make use of subroutines and functions in some form, it might be expected that subroutines and functions would be described earlier, but that is not the case. Chapter 12 describes them in detail. Chapter 11 describes all program units—the main program, modules, external subprograms, and block data program units.

Internal procedures and module procedures gain access to information in their hosts by host association. A USE statement specifying a module can appear in a main program, a subprogram, a module, an interface body, or a block data subprogram to gain access to the module's public information. This method of access is called use association. Association is described in 16.

Figure 2-1 illustrates the organization of a sample Fortran program. The lines with thin arrows represent internal and external subprogram references with the arrow pointing to the subprogram. The thick solid arrows represent access by use association with the arrow pointing to the position of a USE statement.

2.1.1.1 Main Program

The main program is required; if there are other program units, the main program acts as a controller; that is, it takes charge of the program and controls the order in which procedures are executed.

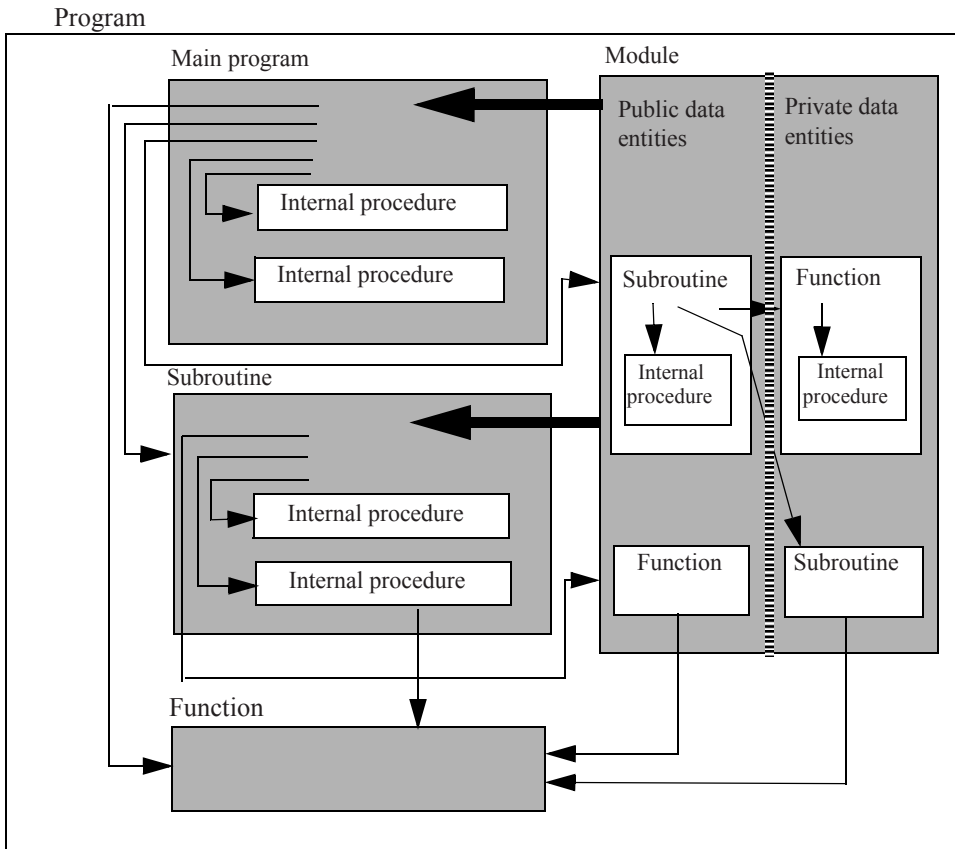


Figure 2-1 Example of program packaging. The thick arrows represent use association; the thin arrows represent procedure references.

2.1.1.2 Module

A module contains definitions that can be made accessible to other program units by use association. These definitions include data definitions, type definitions, definitions of procedures known as module procedures, and specifications of procedure interfaces. A module procedure may be invoked by another module procedure in the module or by other program units that access the module. Fortran 2003 introduced intrinsic modules; there were no intrinsic modules in earlier standards. These are the `ISO_FORTRAN_ENV` module (13.6.1) that provides public entities relating to the environment such as input/output units and storage sizes, the `ISO_C_BINDING` module (15.3) that provides access to named constants representing kind values that are compatible with C types, and three IEEE modules (14.3) that provide support for exceptions and IEEE arithmetic.

2.1.1.3 External Subprogram

An external subprogram (a function or a subroutine) may be used to perform a task or calculation on entities available to the external subprogram. These entities may be the arguments to the subprogram that are provided in the reference, entities defined in the subprogram, or entities accessible from modules or common blocks. A CALL statement is used to invoke a subroutine. A function is invoked when its value is needed in an expression. The computational process that is specified by a function or subroutine subprogram is called a **procedure**. An external subprogram provides one way to define a procedure. It may be invoked from other program units of the Fortran program. Unless it is a pure procedure, a subroutine or function may change the program state by changing the values of data objects accessible to the procedure.

2.1.1.4 Block Data Program Unit

A block data program unit (11.5) contains data definitions only and is used to specify initial values for a restricted set of data objects.

2.1.1.5 Compilation

Prior to the introduction of modules into Fortran, program units could be compiled independently with no need for information from any other program unit. Any information needed in more than one program unit had to be replicated wherever it was needed. The compiled program unit could be used in a number of applications without the necessity of recompiling; this is called **independent compilation**.

If a program unit contains a USE statement, the referenced module must be available in some form when that program unit is compiled.

There are many ways to implement modules; however, most implementations require compilation of modules prior to compilation of any program units that use the modules. The compilation often produces a file containing encoded or summarized information about the module, which is accessed when a program using the module is compiled.

The situation regarding the availability of include files is somewhat similar, but because include files are simply inserted as text in a program, they are not usually preprocessed in any way.

2.1.2 Procedures

A procedure specifies a task or a calculation, usually one that can be separated out from the main flow or one that is needed in different parts of the program. A procedure may take the form of a subroutine or a function. Every procedure has an interface that must be unique in some way. A set of generic procedures may be identified by the same name or symbol, but made unique by their arguments. A procedure may be defined by means other than the Fortran language.

2.1.2.1 Internal Procedures

Main programs, module subprograms, and external subprograms may contain internal subprograms, which may be either subroutines or functions. The procedures they de-

fine are called **internal procedures**. Internal subprograms must not themselves contain internal subprograms, however. The main program, external subprogram, or module subprogram that contains an internal subprogram is referred to as the internal subprogram's **host**. Entities known in a host are available to an internal procedure by host association. Internal procedures may be invoked within their host or within other internal procedures in the same host. Internal procedures are described in 12.

There is also an obsolescent feature, the statement function (12.4.4), which specifies a function by a single statement.

2.1.2.2 Procedure Interfaces

An interface provides the procedure name, the number of arguments, their types, attributes, names, and the type and attributes of a function result. This information is required in some cases, such as for a dummy argument, which assumes the shape of its actual argument (12.5.1.2). The information also allows the processor to check the validity of an invocation.

If a procedure interface is not inherently available, it may be specified in an interface block. All program units, except block data, may contain procedure interface blocks. A procedure interface block contains one or more interface bodies that are used to describe the interfaces of procedures that would otherwise be unknown. Interface blocks are used for external procedures, dummy procedures, procedure pointers, abstract procedures, or type-bound procedures. An interface block with a generic specification may be used to describe generic procedures or user-defined operators, assignment, or input/output. Procedure interfaces are described in 12.

2.1.2.3 Generic Procedures

Fortran has the concept of a generic procedure, that is, one that can accept arguments that have different types in different invocations. If the procedure is a function, in most cases the type of the result is the same as that of the arguments. An example is the intrinsic SIN (the sine function), which can accept a real, double precision, or complex argument. A user-defined procedure also can be generic. A user defines several specific procedures, and either collects their interfaces in an interface block with a generic specification or lists them in a GENERIC statement in the type definition. The identifier that appears in the generic specification or the GENERIC statement may be used to reference the specific procedure whose arguments match those of the reference.

2.1.2.4 Procedures Defined by Other Languages

Chapter 15 describes how procedures defined by means of the C programming language can be accessed from Fortran and how procedures defined in Fortran can be accessed from C programs. Other languages may be accommodated by these same mechanisms. The mechanisms are not limited to C, but are described in terms of C protocols. Some of the additions to Fortran 2003 to facilitate this process are useful in themselves to strictly Fortran programs, such as the VALUE attribute for dummy arguments (5.9.2), enumerations (4.6), and stream input/output (9.1.5.3).

2.2 Data Environment

Before a calculation can be performed, its data environment must be established. The data environment consists of data objects that possess properties, attributes, and values. The steps in a computational process generally specify operations that are performed on operands (or objects) to create desired results or values. Operands may be constants, variables, constructors, function references, or more complicated expressions made up of these items; each operand has a data type (which may be dynamic); it may have type parameters; and, if it is defined, it has a value. A data object has attributes in addition to type. Chapter 4 discusses data type in detail; Chapter 5 discusses how program entities and their attributes are declared; and Chapters 6 and 7 describe how data objects may be used.

2.2.1 Data Type

The Fortran language provides five intrinsic data types—real, integer, complex, logical, and character—and allows users to define additional types. Sometimes it is natural to organize data in combinations consisting of several components of different types. Because the data describe one object, it is convenient to have a means to refer to this aggregation of data by a single name. In Fortran, an aggregation of such data values is called a **structure**. To use a structure, a programmer must first define the type of the structure. Once the new type is defined, any number of structures (or objects) of that type may be declared.

Some applications require related objects, such as a basic line plus a line of a certain style (dotted or dashed), or of a certain color, or both style and color. A base type may be defined and then extended by adding different components. When a type is defined, it is not necessary to specify that it may be extended. Generic procedures may be defined (such as `DRAW` or `ADD_TO_FIGURE`) that accept as an actual argument an object of the base type or any extension of it. Such an argument that may be of any of these types is polymorphic.

2.2.2 Type Parameters

Both intrinsic and user-defined types may have parameters. For the intrinsic types, a kind type parameter specifies a particular representation. In addition, the character type has a length parameter.

Each of the intrinsic types may have more than one representation (specified by a `KIND` parameter). The Fortran standard requires at least two different representations for each of the real and complex types that correspond to “single precision” and “double precision”, and permits more.

A type parameter for a user-defined type is also either a kind type parameter or a length type parameter. Type parameters for user-defined types are specified in the type definition.

Portable mechanisms for specifying precision are provided so that numerical algorithms that depend on a minimum numeric precision can be programmed to produce reliable results regardless of the processor’s characteristics. Fortran permits more than one representation for the integer, logical, and character types as well. Alternative rep-

representations for the integer type permit different ranges of integers. Alternative representations for the logical type might include a “packed logical” type to conserve memory space and an “unpacked logical” type to increase speed of access. The large number of characters required for ideographic languages, such as those used in Asia with thousands of different graphical symbols, cannot be represented as concisely as alphabetic characters and require “more precision”. For international usage Fortran 2003 encourages support of the ISO 10646 character set (1.5).

A kind type parameter value must be known at compile time and may be used to resolve generic procedure references. A length type parameter value need not be known at compile time; it may be used for character lengths, array dimensions, or other sizes. If it is a deferred type parameter, indicated by a colon (:), it may change during execution. If it is an assumed type parameter, indicated by an asterisk (*), it assumes its value from another entity, such as an actual argument.

Examples of type declarations with parameters are:

```
complex (kind = HIGH) x
integer (kind = SHORT) days_of_week
character (kind = ISO_10646, len = 500) HAIKU
type MY_ARRAY (pick_kind, rows, cols)          ! Type definition
    integer, kind :: pick_kind
    integer, len :: rows, cols
    real (pick_kind) :: VALUES (rows, cols)
end type MY_ARRAY
type(MY_ARRAY) AA(HIGH, i, j)
```

where HIGH, SHORT, and ISO_10646 are named integer constants given appropriate values by the programmer. The length parameter for the character string HAIKU has the value 500. AA is of type MY_ARRAY; its single component, VALUES, is a real array of kind HIGH and dimension (i, j), where i and j are specification expressions.

2.2.3 Dimensionality

Single objects, whether intrinsic or user-defined, are scalar. Even though a structure has components, it is technically a scalar. A set of scalar objects, all of the same type, may be arranged in patterns involving columns, rows, planes, and higher-dimensioned configurations to form arrays. It is possible to have arrays of structures. An array may have up to seven dimensions. The number of dimensions is called the **rank** of the array. It is declared when the array is declared and cannot change. The **size** of the array is the total number of elements and is equal to the product of the extents in each dimension. The **shape** of an array is the list of its extents. Two arrays that have the same shape are said to be **conformable**. A scalar is conformable with any array. Examples of array declarations are:

```
real :: coordinates (100, 100)
integer :: distances (50)
type(line) :: mondrian(10)
```


An array is an object and may appear in an expression or be returned as a function result. Intrinsic operations involving arrays of the same shape are performed element-by-element to produce an array result of the same shape. There is no implied order in which the element-by-element operations are performed.

A portion of an array, such as an element or section, may be referenced as a data object. An array element is a single element of the array and is scalar. An array section is a subset of the elements of the array and is itself an array.

2.2.4 Dynamic Data

Data objects may be dynamic in size, shape, type, or length type parameters, but not rank or kind type parameters. The dynamic data objects are:

- polymorphic objects
- pointers
- allocatable objects
- automatic objects

The type of a polymorphic object (5.2) may change during program execution. Objects that may have both a dynamic type as well as a dynamic size and shape are data pointers, allocatable variables, and dummy arguments. Automatic objects appear in subprograms and come into existence when the subprogram is invoked.

Dynamic type was introduced in Fortran 2003. An entity that is not polymorphic has both a declared and a dynamic type, but they are the same. The dynamic type of a polymorphic object that is not allocated (6.7.1) or associated (7.5.5.1) is its declared type. The CLASS keyword is used to declare polymorphic entities. An object declared with CLASS (*) is an unlimited polymorphic object with no declared type.

Procedures and data objects in Fortran may be declared to have the POINTER attribute. A procedure pointer must be a procedure entity. A data pointer must be associated with a target before it can be used in any calculation. This is accomplished by allocation (6.7.1.2) of the space for the target or by assignment of the pointer to an existing target (7.5.5.1). A pointer assignment statement is provided to associate a pointer with a target (declared or allocated). It makes use of the symbol pair => rather than the single character =; otherwise, it is executed in the same way that an ordinary assignment statement is executed, except that instead of assigning a value it associates a pointer with a target. For example,

```
real, target :: VECTOR(100)
real, pointer :: ODDS(:)
. . .
ODDS => VECTOR(1:100:2)
```

The pointer assignment statement associates ODDS with the odd elements of VECTOR. The assignment statement

```
ODDS=1.5
```

defines each odd element of VECTOR with the value 1.5. Later in the execution sequence, the pointer ODDS could become associated with a different target by pointer assignment or allocation, as long as the target is a one-dimensional, default real array. Chapter 7 describes the pointer assignment statement.

If a pointer object is declared to be an array, its size and shape may change dynamically, but its rank is fixed by the declaration. If a pointer target is polymorphic, the pointer must be of a type that is compatible with the target, or both the pointer and target must be declared unlimited polymorphic. An example of pointer array declaration and allocation is:

```
real, pointer :: lengths (:)  
allocate (lengths (200))
```

A variable may be declared to have the ALLOCATABLE attribute. Space must be allocated for the variable before it can be used in any calculation. The variable may be deallocated and reallocated with a different type, length type parameters, and shape as the program executes. As with a pointer, the rank is fixed by the declaration. An allocatable variable cannot be made to point to an existing named object; the object always must be created by an ALLOCATE statement. An example of allocatable array declaration and allocation is:

```
real, allocatable :: lengths (:)  
allocate (lengths (200))
```

The similarities of these examples reflect the similarity of some of the uses of allocatable arrays and pointers, but there are differences. Pointers may be used to create dynamic data structures, such as linked lists and trees. The target of a pointer can be changed by reallocation or pointer assignment; the new target must be of the same rank but may have different extents in each dimension. The attributes of an allocatable variable can be changed only by deallocating and reallocating the variable. There is a MOVE_ALLOC intrinsic function that can be used if the values of the elements of an allocatable array are to be preserved when its size is changed. Use of allocatable variables generally leads to more efficient execution than use of the more flexible pointers.

Only pointers and allocatable objects may be allocated or deallocated. It is possible to inquire whether an object is currently allocated. Chapter 5 describes the declaration of pointers and allocatable objects; Chapter 6 covers the ALLOCATE and DEALLOCATE statements; Chapter 13 and Appendix A describe the ASSOCIATED intrinsic inquiry function for pointers and the ALLOCATED intrinsic inquiry function for allocatable variables. Chapter 15 describes dynamic interoperable objects.

Automatic data objects, either arrays or character strings (or both), may be declared in a subprogram. These local data objects are created on entry to the subprogram and disappear when the execution of the subprogram completes. These are useful in subprograms for temporary arrays and character strings whose sizes are different for each reference to the subprogram. An example of a subprogram unit with an automatic array TEMP is:

```
subroutine SWAP_ARRAYS (A, B)
  real, dimension (: ) :: A, B
  real, dimension (size (A)) :: TEMP

  TEMP = A
  A = B
  B = TEMP
end subroutine SWAP_ARRAYS
```

A and B are assumed-shape array arguments; that is, they take on the shape of the actual arguments. TEMP is an automatic array that is created the same size as A on entry to subroutine SWAP_ARRAYS. SIZE is an intrinsic function.

2.2.5 Packaging and Encapsulation

The packaging of a fair-sized program is an important design consideration when a new Fortran application is planned. The most important benefit of packaging is information hiding. Entities can be kept inaccessible except where they are actually needed. This provides some protection against inadvertent misuse or corruption, thereby improving program reliability. Packaging can make the logical structure of a program more apparent by hiding complex details at lower levels. Programs are therefore easier to comprehend and less costly to maintain. The Fortran features that provide these benefits are

- user-defined types
- internal procedures
- modules

The accessibility of a user-defined type in a module may be public, private, or protected. In addition, even if the type is public, it may have private components. A type definition has a type-bound procedure part in which the procedures bound to that type are specified.

Internal procedures may appear in main programs, module subprograms, and external subprograms; they are known only within their host. The name of an internal procedure must not be passed as an argument. The Fortran standard further restricts internal procedures in that an internal procedure must not itself be the host of another internal procedure. However, statement functions may appear within an internal procedure.

Modules provide the most comprehensive opportunities to apply packaging concepts including several levels of organization and hiding (5.8). The entities specified in a module (types, data objects, procedures, interfaces, etc.) may be made available to other scoping units; may be made available, but protected from corruption outside the module; or may be kept private to the module. Thus modules provide flexible encapsulation facilities for entities in an object-oriented application. The procedures, mentioned in a type definition (4.4.2) and referred to as type-bound procedures (4.4.11), generally appear as module procedures in the module that contains the type definition.

In addition to the usual capabilities of procedures, these type-bound procedures may specify

- defined operators
- defined assignment
- defined input/output
- finalization

Finalization is accomplished by a final procedure that is invoked automatically just before an object of the type is destroyed by deallocation, the execution of a RETURN or END statement, or some other means.

Of course, more than one type definition may appear in a module, so if there is a need for communication among separate but related objects, the module provides the appropriate means for permitting and controlling access to information.

2.3 Program Execution

During program execution, constructs and statements are executed in a prescribed order. Variables become defined with values and may be redefined later in the execution sequence. Procedures are invoked, perhaps recursively. Space may be allocated and later deallocated. The targets of pointers may change. The types of polymorphic variables may change.

2.3.1 Execution Sequence

Program execution begins with the first executable construct in the main program. An executable construct is an instruction to perform one or more of the computational actions that determine the behavior of the program or control the flow of the execution of the program. These actions include performing arithmetic, comparing values, branching to another construct or statement in the program, invoking a procedure, or reading from or writing to a file or device. Examples of executable statements are:

```

      read (5, *) z, y
      x = (4.0 * z) + base
      if (x > y) go to 100
      call calculate (x)
100  y = y + 1

```

When a procedure is invoked, its execution begins with the first executable construct after the entry point in the procedure. On normal return from a procedure invocation, execution continues where it left off in the invoking procedure.

Unless a control statement or construct that alters the flow of execution is encountered, program statements are executed in the order in which they appear in a program unit until a STOP, RETURN, or END statement is executed. Branch statements specify a change in the execution sequence and consist of the various forms of GO TO statements, a procedure reference with alternative return specifiers, EXIT and CYCLE state-

ments in DO constructs, and input/output statements with branch label specifiers, such as ERR, END, and EOR specifiers. The control constructs (IF, CASE, DO, and SELECT TYPE) can cause internal branching implicitly within the structure of the construct. The SELECT TYPE construct chooses a block of code based on the dynamic type of its polymorphic selector. Chapter 8 discusses in detail control flow within a program.

Another feature of Fortran 2003 is asynchronous input/output. It allows computation to occur in parallel with an input/output process if the processor supports parallel processing. A WAIT statement may be used to synchronize the processes. This and other new input/output features are described in 9.

Normal termination of execution occurs if the END statement of a main program or a STOP statement is executed. Normal termination of execution also may occur in a procedure defined by means other than Fortran. If a Fortran program includes procedures executed by a companion processor, the normal termination process will include the effect of executing the C exit function.

2.3.2 Definition and Undefined

Unless initialized, variables have no value initially; uninitialized variables are considered to be **undefined**. Variables may be initialized in type declaration statements, type declarations, DATA statements, or by means other than Fortran; initialized variables are considered to be **defined**. Some variables initialized by default initialization, such as that specified in a type definition, are initialized when the variables come into existence, whereas other variables such as those initialized in a DATA statement are initialized when execution begins.

A variable may acquire a value or change its current value, typically by the execution of an assignment statement or an input statement. Thus it may assume different values at different times, and under some circumstances it may become undefined. This is part of the dynamic behavior of program execution. Defined and undefined are the Fortran terms that are used to specify the definition status of a variable. The events that cause variables to become defined and undefined are described in 16.

A variable is considered to be defined only if all parts of it are defined. For example, all the elements of an array, all the components of a structure, or all characters of a character string must be defined; otherwise, the array, structure, or string is undefined. Fortran permits zero-sized arrays and zero-length strings; these are always considered to be defined.

Pointers have both a definition status and an association status. When execution begins, the association status of all pointers is undefined, except for data or default initialized pointers given the disassociated status. During execution a pointer may become disassociated, or it may become associated with a target. At some point the association status may revert to undefined. Even when the association status of a pointer is defined, the pointer is not considered to be defined unless the target with which it is associated is defined. Pointer targets become defined in the same way that any other variable becomes defined, typically by the execution of an assignment or input statement.

Allocatable variables have a definition status and an allocation status. The allocation status is never undefined.

2.3.3 Scope

The scope of a program entity is the part of the program in which that entity is known, is available, and can be used. A scoping unit is

1. a program unit or subprogram, excluding any scoping units in it
2. a derived-type definition
3. an interface body, excluding any scoping units in it

Some entities have scopes that are something other than a scoping unit. For example, the scope of a name, such as a variable name, can be any of the following:

1. an executable program
2. a scoping unit
3. a construct
4. a statement or part of a statement

The scope of a label is a scoping unit. The scope of an input/output unit is a program.

2.3.4 Association

Association is the concept that is used to describe how different entities in the same scoping unit or different scoping units can share values and other properties. Argument association allows values to be shared between a procedure and the program that calls it. Use association and host association allow entities described in one part of a program to be used in another part of the program. Use association makes entities defined in modules accessible, and host association makes entities in the containing environment available to a contained procedure. The `IMPORT` statement (12.5.2), introduced in Fortran 2003, makes entities in a host scoping unit available in an interface body by host association.

Additional forms of association are inheritance association (between the entities in an extended type and its parent), linkage association (between corresponding Fortran and C entities), and construct association (relevant to the `ASSOCIATE` and `SELECT TYPE` constructs). The complete description of association may be found in 16.

An old form of association, storage association, which allows two or more variables to share storage, can be set up by the use of `EQUIVALENCE`, `COMMON`, or `ENTRY` statements. It is best avoided.

2.4 Terms

Frequently used Fortran terms are defined in this section. Definitions of less frequently used terms may be found by referencing the index of this handbook or Annex A of the Fortran 2003 standard.

Entity	This is the general term used to refer to any Fortran “thing”, for example, a program unit, a procedure, a common block, a variable, an expression value, a constant, a statement label, a construct, an operator, an interface, a type, an input/output unit, a namelist group, etc.
Name	A name is used to identify many different entities of a program such as a program unit, a named variable, a named constant, a common block, a construct, a formal argument of a subprogram (dummy argument), or a user-defined type (derived type). The rules for constructing names are given in 3.
Named entity	A named entity is referenced by a name without any qualification such as an appended subscript list or substring range.
Data object	A data object is a constant, a variable, or a subobject of a constant. It may be a scalar or an array. It may be of intrinsic or derived type.
Constant	A constant is a data object whose value cannot be changed. A named entity with the PARAMETER attribute is called a named constant . A constant without a name is called a literal constant . A constant may be a scalar or an array.
Variable	A variable is a data object whose value can be defined and re-defined. A variable may be a scalar or an array.
Local variable	A variable that is in a main program, module, or subprogram and is not associated by being: a dummy argument, in COMMON, a BIND(C) variable, or accessed via host or USE association. A subobject of a local variable is also a local variable.
Subobject of a constant	A subobject of a constant is a portion of a constant. The portion referenced may depend on the value of a variable, in which case it is neither a constant nor a variable.
Data entity	A data entity is a data object or the result of the evaluation of an expression. A data entity has a type, possibly type parameters, and a rank (a scalar has rank zero). It may have a value.
Expression	An expression may be a simple data reference or it may specify a computation and thus be made up of operands, operators, and parentheses. The type, type parameters, value, and rank of an expression result are determined by the rules in 7.
Function reference	A function reference invokes a function. It is made up of the name of a function followed by a parenthesized list of arguments, which may be empty. The type, type parameters,

and rank of the result are determined by the interface of the function and the reference.

Data type	A data type provides a means for categorizing data. Each intrinsic and user-defined data type has—a name, a set of values, a set of operators, and a means to represent values of the type in a program. For each data type there is a type specifier that is used to declare objects of the type.
Type parameter	There are two categories of type parameters for types: kind and length. For intrinsic types a kind type parameter indicates the range for the integer type, the decimal precision and exponent range for the real type and parts of the complex type, and the machine representation method for the character and logical types. The length type parameter indicates a length for the intrinsic character type. For a derived type, the type parameters are defined in its type definition.
Derived type	A derived type (or user-defined type) is a type that is not intrinsic; it requires a type definition to name the type and specify its parameters and components. The components may be of intrinsic or user-defined types. An object of derived type is called a structure. For each derived type, a structure constructor is available to specify values. Operations on objects of derived type must be defined by a function. Assignment for derived-type objects is defined intrinsically, but may be redefined by a subroutine. Finalizers may be specified for derived-type objects. Data entities of derived type may be used as procedure arguments and function results, and may appear in input/output lists and other places. Derived types may be extended by inheritance.
Ultimate component	The ultimate components of a derived type entity are the lowest-level components that have storage in the entity. They are a) any components that are of an intrinsic type, b) any components that have the <code>ALLOCATABLE</code> or <code>POINTER</code> attribute (the entity has storage for the pointer or allocation descriptor, but the object or target does not, itself, have storage in the entity), and c) the ultimate components of any derived type components that have neither the <code>ALLOCATABLE</code> nor <code>POINTER</code> attribute. The ultimate components are subject to, for example, storage association rules. There is a distinction between a component of derived type and an allocatable or pointer component of the same type. In the first case, the elements of the derived type component are

	ultimate components; in the other cases only the descriptor or pointer is an ultimate component
Inheritance	Inheritance is the process of automatically acquiring entities (parameters, components, or procedure bindings) from a parent.
Polymorphism	Polymorphism is the ability to change type during program execution. Dummy arguments, pointers, and allocatable objects may be polymorphic.
Scalar	A scalar is a single object of any intrinsic or derived type. A structure is scalar even if it has a component that is an array. The rank of a scalar is zero.
Array	An array is an object with the dimension attribute. It is a collected set of scalar data, all of the same type and type parameters. The rank of an array is at least one and at most seven. An array of any rank may be of zero size. An array of size zero or one is not a scalar. Data entities that are arrays may be used as expression operands, procedure arguments, and function results, and may appear in input/output lists, as well as other places.
Subobject	A subobject is a portion of a data object. Portions of a data object may be referenced and defined (if the object is a variable) separately from other portions of the object. Array elements and array section are portions of arrays. Substrings are portions of character strings. Structure components are portions of structures. Portions of complex objects are the real and imaginary parts. Subobjects are referenced by designators or intrinsic functions and are considered to be data objects themselves.
Designator	Sometimes it is convenient to reference only part of an object, such as an element or section of an array, a substring of a character string, or a component of a structure. This requires the use of a designator which is the name of the object followed by zero or more selectors that select a part of the object.
Selector	This term is used in several different ways. A selector may designate part of an object (array element, array section, substring, or structure component) or the set of values for which a CASE block is executed, or the dynamic type for which a SELECT TYPE block is executed, or the object associated with the name in an ASSOCIATE construct.
Declaration	A declaration is a nonexecutable statement that specifies the attributes of a program element. For example, it may be used to specify the type of a variable or function or the shape of an ar-

ray. It may indicate that an entity is a data pointer or a procedure pointer. Attributes that were introduced in Fortran 2003 are: `ASYNCHRONOUS`, which indicates that the value of the variable may change outside the execution flow due to a possibly simultaneous input/output process; `BIND (C)`, which is used to indicate data and functions that interoperate with C; `PROTECTED`, which prohibits any change to the value of the variable or the association status of the pointer outside the module in which it is declared; `VALUE`, which, when applied to a dummy argument, specifies an argument passing mechanism useful in C interoperability; and `VOLATILE`, which indicates that the value of the variable may change by means other than the normal execution sequence

Definition

This term is used in several ways. A data object is said to be defined when it has a valid or predictable value; otherwise it is undefined. It may be given a valid value by execution of statements such as assignment or input. Under certain circumstances described in 16, it may subsequently become undefined.

Procedures and derived types are said to be defined when their descriptions have been supplied by the programmer and are available in a program unit.

The association status of a pointer is defined when the pointer is associated or disassociated; otherwise, it is undefined.

Statement keyword

A statement keyword is part of the syntax of a statement. Each statement, other than an assignment statement, pointer assignment statement, or statement function definition, begins with a statement keyword. Some statement keywords appear in internal positions within statements. Examples of these keywords are `THEN`, `KIND`, and `INTEGER`. Statement keywords are not reserved; they may be used as names.

List keyword

A list keyword is a name that is used to identify an item in a list (rather than its position) such as an argument list, type parameter list, or structure constructor list. Keywords for the argument lists of all of the intrinsic procedures are specified by the standard (A). Keywords for user-supplied external procedures may be specified in a procedure interface block. Keywords for structure constructors and user-defined type parameters are specified in the type definition.

Sequence

A sequence is a set ordered by a one-to-one correspondence with the numbers 1, 2, through n . The number of elements in

the sequence is n . A sequence may be empty, in which case it contains no elements.

Operator	An operator indicates a computation involving one or two operands. Fortran defines a number of intrinsic operators; for example, <code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>**</code> with numeric operands, and <code>.NOT.</code> , <code>.AND.</code> , <code>.OR.</code> with logical operands. In addition, users may define operators for use with operands of intrinsic or derived types.
Construct	A construct is a sequence of statements starting with an <code>ASSOCIATE</code> , <code>DO</code> , <code>FORALL</code> , <code>IF</code> , <code>SELECT CASE</code> , <code>SELECT TYPE</code> , or <code>WHERE</code> statement and ending with the corresponding terminal statement.
Executable construct	An executable construct is an action statement (such as a <code>READ</code> statement) or a construct (such as a <code>DO</code> or <code>CASE</code> construct).
Procedure	A procedure is defined by a sequence of statements that expresses a computation that may be invoked as a subroutine or function during program execution. It may be an intrinsic procedure, an external procedure, an internal procedure, a module procedure, a dummy procedure, or a statement function. If a subprogram contains an <code>ENTRY</code> statement, it defines more than one procedure.
Procedure interface	A procedure interface is a sequence of statements that specifies the name and characteristics of a procedure, the name and attributes of each dummy argument, and the generic specifier by which it may be referenced, if any.
Reference	A data object reference is the appearance of the object designator in a statement requiring the value of the object. A procedure reference is the appearance of the procedure designator, operator symbol, or assignment symbol in an executable program requiring execution of the procedure. A module reference is the appearance of the module name in a <code>USE</code> statement.
Intrinsic	Anything that is defined by the Fortran processor is intrinsic. There are intrinsic data types, procedures, modules, operators, and assignment. These may be used freely in any scoping unit. The Fortran programmer may define types, procedures, modules, operators, and assignment; these entities are not intrinsic.

Companion processor	A companion processor is a processor that provides mechanisms by which global data and procedures may be referenced or defined—perhaps by means other than Fortran, such as the C programming language.
Scoping unit	A scoping unit is a portion of a program in which a name has a fixed meaning. A program unit or subprogram generally defines a scoping unit. Type definitions and procedure interface bodies also constitute scoping units. Scoping units are non-overlapping, although one scoping unit may contain another in the sense that it surrounds it. If a scoping unit contains another scoping unit, the outer scoping unit is referred to as the host scoping unit of the inner scoping unit.
Association	In general, association permits an entity to be referenced by different names in a scoping unit or by the same or different names in different scoping units. There are several kinds of association: the major ones are name association, pointer association, inheritance association, and storage association. Name association is argument association, use association, host association, linkage association, and construct association.
Inheritance association	Inheritance association occurs between the inherited entities of an extended type and the corresponding entities of its parent.
Linkage association	Linkage association occurs between a module variable with the BIND(C) attribute and the relevant C variable or between a Fortran common block and the relevant C variable. It has the scope of the program.
Construct association	Construct association occurs between the selector in an ASSOCIATE or SELECT TYPE construct and the associate name of the construct. It has the scope of the construct.

2.5 High-Level Syntax Forms

The form of a program (R201) is:

```

program-unit
[ program-unit ] . . .

```

The forms for Fortran program units are shown in the first section below. The constructs that may appear in a program unit are shown in the subsequent sections. All program units may have a specification part. The main program and the three forms of subprogram (module, external, and internal) may have an execution part.

The notation used in this chapter is the same as that used to show the syntax in all the remaining chapters; it is described in 1.3 along with an assumed syntax rule and

some frequently used abbreviations for syntax terms. This is not the complete set of rules; many lower-level rules are missing. Many of these rules may be found in the following chapters. The Fortran 2003 standard [7] contains the complete syntax rules.

2.5.1 Fortran Program Units

The forms of a program unit (R202) are:

```
main-program
module
external-subprogram
block-data
```

The form of a main program (R1101) is:

```
[ PROGRAM program-name ]
  [ specification-part ]
  [ execution-part ]
[ CONTAINS
  internal-subprogram
  [ internal-subprogram ] ... ]
END [ PROGRAM [ program-name ] ]
```

The form of a module (R1104) is:

```
MODULE module-name
  [ specification-part ]
[ CONTAINS
  module-subprogram
  [ module-subprogram ] ... ]
END [ MODULE [ module-name ] ]
```

The form of a module subprogram (R1108) and an external subprogram (R203) is:

```
subprogram-heading
  [ specification-part ]
  [ execution-part ]
[ CONTAINS
  internal-subprogram
  [ internal-subprogram ] ... ]
subprogram-ending
```

The form of an internal subprogram (R211) is:

```
subprogram-heading
  [ specification-part ]
  [ execution-part ]
subprogram-ending
```

The forms of a subprogram heading (R1224, R1232) are:

```
[ prefix ] [ declaration-type-spec ] FUNCTION function-name &
    ( [ dummy-argument-list ] ) [ suffix ]
[ prefix ] SUBROUTINE subroutine-name [ ( [dummy-argument-list ] ) ] [ binding-spec ]
```

A prefix (R1228) is any combination of the keywords:

```
RECURSIVE
PURE
ELEMENTAL
```

A suffix (R1229) is one of the forms:

```
RESULT ( result-name ) [ binding-spec ]
binding-spec [ RESULT ( result-name ) ]
```

A binding specification (R509) is:

```
BIND ( C [ , NAME = scalar-char-initialization-expr ] )
```

The forms of a subprogram ending (R1230, R1234) are:

```
END [ FUNCTION [ function-name ] ]
END [ SUBROUTINE [ subroutine-name ] ]
```

The form of a block data program unit (R1116) is:

```
BLOCK DATA [ block-data-name ]
    [ specification-part ]
END [ BLOCK DATA [ block-data-name ] ]
```

2.5.2 The Specification Part

The form of the specification part (R204) is:

```
[ use-statement ] ...
IMPORT [ [ :: ] import-name-list ] ...
[ implicit-part ]
[ declaration-construct ] ...
```

The forms of a USE statement (R1109) are:

```
USE [ [ , module-nature ] :: ] module-name [ , rename-list ]
USE [ [ , module-nature ] :: ] module-name , ONLY : [ only-list ]
```

The form of the implicit part (R206) is:

```
[ implicit-part-statement ] ...
IMPLICIT implicit-spec-list
```

The forms of an implicit part statement (R205) are:


```

IMPLICIT implicit-spec-list
PARAMETER ( named-constant = initialization-expr &
           [ , named-constant = initialization-expr ] ... )
entry-statement
format-statement

```

The forms of an implicit specification (R550) are:

```

NONE
declaration-type-spec ( letter-spec-list )

```

The forms of a declaration construct (R207) are:

```

declaration-type-spec [ [ , attribute-spec ] ... :: ] entity-declaration-list
specification-statement
derived-type-definition
interface-block
enumeration-definition
entry-statement
format-statement
statement-function-statement

```

The forms of a declaration type specification (R502) are:

```

INTEGER [ kind-selector ]
REAL [ kind-selector ]
DOUBLE PRECISION
COMPLEX [ kind-selector ]
CHARACTER [ character-selector ]
LOGICAL [ kind-selector ]
TYPE ( derived-type-spec )
CLASS ( derived-type-spec )
CLASS ( * )

```

The form of a kind selector (R404) is:

```

( [ KIND = ] kind-value )

```

The forms of a character selector (R424) are:

```

( length-value [ , [ KIND = ] kind-value ] )
( LEN = length-value [ , KIND = kind-value ] )
( KIND = kind-value [ , LEN = length-value ] )
* character-length [ , ]

```

A length value (R402) has one of the forms:

```

scalar-integer-expression
*
:

```

A kind value (R404) has the from:

scalar-integer-initialization-expr

A character length (R426) has one of the forms:

(length-value)
scalar-integer-literal-constant

A derived-type specification (R455) has the from:

type-name [(type-parameter-spec-list)]

A type parameter specification (R456) has the from:

[keyword =] length-value

The forms of an attribute specification (R503) are:

ALLOCATABLE
ASYNCHRONOUS
BIND (C [, NAME = scalar-char-initialization-expr])
DIMENSION (array-spec)
EXTERNAL
INTENT (intent-spec)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PROTECTED
PUBLIC
SAVE
TARGET
VALUE
VOLATILE

The form of an entity declaration (R504) is:

object-name [(array-spec)] [* character-length] [initialization]

The forms of initialization (R506) are:

= initialization-expr
=> function-reference

The forms of specification statements (R212) are:

ALLOCATABLE [::] object-name [(deferred-shape-spec-list)] &
[, object-name [(deferred-shape-spec-list)]] ...
ASYNCHRONOUS [[::] variable-name-list]
BIND (C [, NAME = scalar-char-initialization-expr]) [::] bind-entity-list

```

COMMON [ / [ common-block-name ] / ] common-block-object-list
DATA data-statement-object-list / data-value-list / &
    [ [ , ] data-statement-object-list / data-value-list / ]
DIMENSION [ :: ] array-name ( array-spec ) [ , array-name ( array-spec ) ] ...
EQUIVALENCE equivalence-set-list
EXTERNAL [ :: ] external-name-list
INTENT ( intent-spec ) [ :: ] dummy-argument-name-list
INTRINSIC [ :: ] intrinsic-procedure-name-list
NAMelist / namelist-group-name / namelist-group-object-list
OPTIONAL [ :: ] dummy-argument-name-list
POINTER [ :: ] pointer-declaration-list
PARAMETER ( named-constant = initialization-expr &
    [ , named-constant = initialization-expr ] ... )
PROCEDURE ( [ procedure-interface ] ) [ [ , procedure-attribute-spec ] ... :: ] &
    procedure-declaration-list
PROTECTED [ :: ] entity-name-list
PUBLIC [ [ :: ] access-id-list ]
PRIVATE [ [ :: ] access-id-list ]
SAVE [ [ :: ] saved-entity-list ]
TARGET [ :: ] object-name [ ( array-spec ) ] [ , object-name [ ( array-spec ) ] ] ...
VALUE [ :: ] dummy-argument-name-list
VOLATILE [ :: ] variable-name-list

```

The forms of a procedure interface (R1212) are:

```

interface-name
declaration-type-spec

```

The forms of a procedure attribute specification (R1213) are:

```

BIND ( C [ , NAME = scalar-char-initialization-expr ] )
INTENT ( intent-spec )
OPTIONAL
POINTER
PRIVATE
PUBLIC
SAVE

```

The form of a procedure declaration (R1214) is:

```

procedure-entity-name [ => function-reference ]

```

The form of a derived-type definition (R429) is:

```

TYPE [ [ , type-attribute-list ] :: ] type-name [ ( type-parameter-name-list ) ]
    [ type-parameter-definition-statement ] ...
    [ private-or-sequence-statement ]
    [ component-definition-statement ] ...
[ CONTAINS

```

```

    [ PRIVATE ]
    procedure-binding-statement
    [ procedure-binding-statement ] ... ]
END TYPE [ type-name ]

```

The forms of a type attribute (R431) are:

```

ABSTRACT
BIND ( C )
EXTENDS ( parent-type-name )
PRIVATE
PUBLIC

```

The form of a type parameter definition statement (R435) is:

```

INTEGER [ kind-selector ] , type-parameter-attribute-spec :: &
    type-parameter-declaration-list

```

The forms of a type parameter attribute specification (R437) are:

```

KIND
LEN

```

The form of a type parameter declaration (R436) is:

```

type-param-name [ = scalar-integer-initialization-expr ]

```

The forms of a component definition statement (R439) are:

```

declaration-type-spec [ [ , component-attribute-spec-list ] :: ] &
    component-declaration-list
PROCEDURE ( [ procedure-interface ] ) , procedure-component-attribute-spec-list :: &
    procedure-declaration-list

```

The forms of a component attribute specification (R441) are:

```

ALLOCATABLE
DIMENSION ( component-array-spec )
POINTER
PRIVATE
PUBLIC

```

The form of a component declaration (R442) is:

```

component-name [ ( component-array-spec ) ] [ * character-length ] [ initialization ]

```

The forms of a procedure component attribute specification (R446) are:

```

NOPASS
PASS [ ( argument-name ) ]
POINTER
PRIVATE
PUBLIC

```

The form of an interface block (R1201) is:

```
[ ABSTRACT ] INTERFACE [ generic-spec ]
  [ subprogram-heading
    [ specification-part ]
    subprogram-ending ] ...
  [ [ MODULE ] PROCEDURE procedure-name-list ] ...
END INTERFACE [ generic-spec ]
```

The forms of a generic specification (R1207) are:

```
generic-name
OPERATOR ( defined-operator )
ASSIGNMENT ( = )
derived-type-io-generic-spec
```

The form of an enumeration definition (R460) is:

```
ENUM , BIND ( C )
  ENUMERATOR [ :: ] enumerator-list
  [ ENUMERATOR [ :: ] enumerator-list ] . . .
END ENUM
```

2.5.3 The Execution Part

The form of the execution part (R208) is:

```
execution-part-construct
[ execution-part-construct ] ...
```

The forms of an execution part construct (R209) are:

```
executable-construct
entry-statement
format-statement
```

The forms of an executable construct (R213) are:

```
action-statement
associate-construct
case-construct
do-construct
forall-construct
if-construct
select-type-construct
where-construct
```

The forms of an action statement (R214) are:

```
variable = expression
data-pointer-object [ ( bounds-list ) ] => data-target
data-pointer-object ( bounds-remap-list ) => data-target
```

```

procedure-pointer-object => procedure-target
ALLOCATE [ declaration-type-spec : : ] ( allocation-list [ , allocate-option-list ] )
BACKSPACE scalar-integer-expression
BACKSPACE ( position-spec-list )
CALL subroutine-name [ ( [ actual-argument-spec-list ] ) ]
CLOSE ( close-spec-list )
CONTINUE
CYCLE [ do-construct-name ]
DEALLOCATE ( allocate-object-list [ , deallocate-option-list ] )
ENDFILE scalar-integer-expression
ENDFILE ( position-spec-list )
EXIT [ do-construct-name ]
FLUSH scalar-integer-expression
FLUSH ( flush-spec-list )
FORALL ( forall-triplet-specification-list [ , scalar-logical-expression ] ) &
    forall-assignment-statement
GO TO label
GO TO ( label-list ) [ , ] scalar-integer-expression
IF ( scalar-logical-expression ) action-statement
IF ( scalar-numeric-expression ) label , label , label
INQUIRE ( inquire-spec-list )
INQUIRE ( IOLENGTH = scalar-integer-variable ) output-item-list
NULLIFY ( pointer-object-list )
OPEN ( connection-spec-list )
PRINT format [ , output-item-list ]
READ ( io-control-spec-list ) [ input-item-list ]
READ format [ , input-item-list ]
RETURN [ scalar-integer-expression ]
REWIND scalar-integer-expression
REWIND ( position-spec-list )
STOP [ scalar-character-constant ]
STOP digit [ digit [ digit [ digit ] ] ] ]
WAIT ( wait-spec-list )
WHERE ( logical-expression ) where-assignment-statement
WRITE ( io-control-spec-list ) [ output-item-list ]

```

The form of the ASSOCIATE construct (R816) is:

```

[ associate-construct-name : ] ASSOCIATE ( association-list )
    block
END ASSOCIATE [ associate-construct-name ]

```

The form of the CASE construct (R808) is:

```

[ case-construct-name : ] SELECT CASE ( case-expression )
[ CASE ( case-value-range-list ) [ case-construct-name ]
    block ] ...

```

```

[ CASE DEFAULT [ case-construct-name ]
  block ]
END SELECT [ case-construct-name ]

```

The form of the DO construct (R825) is:

```

[ do-construct-name : ] DO [ label ] [ loop-control ]
  block
[ label ] END DO [ do-construct-name ]

```

The form of the FORALL construct (R752) is:

```

[ forall-construct-name : ] &
  FORALL ( forall-triplet-spec-list [ , scalar-logical-expression ] )
  [ forall-body-construct ] ...
END FORALL [ forall-construct-name ]

```

The form of the IF construct (R802) is:

```

[ if-construct-name : ] IF ( scalar-logical-expression ) THEN
  block
[ ELSE IF ( scalar-logical-expression ) THEN [ if-construct-name ]
  block ] ...
[ ELSE [ if-construct-name ]
  block ]
END IF [ if-construct-name ]

```

The form of the SELECT TYPE construct (R821) is:

```

[ select-construct-name : ] SELECT TYPE ( [ associate-name => ] selector )
  [ type-guard [ select-construct-name ]
  block ] . . .
END SELECT [ select-construct-name ]

```

The form of the WHERE construct (R744) is:

```

[ where-construct-name : ] WHERE ( logical-expression )
  [ where-body-construct ] ...
[ ELSEWHERE ( logical-expression ) [ where-construct-name ]
  [ where-body-construct ] ... ] ...
[ ELSEWHERE [ where-construct-name ]
  [ where-body-construct ] ... ]
END WHERE [ where-construct-name ]

```

2.6 Ordering Requirements

Within program units, subprograms, and interface bodies there are ordering requirements for statements and constructs. The syntax rules above do not fully describe the

ordering requirements. Therefore, they are illustrated in Table 2-1. In general, data declarations and specifications must precede executable constructs and statements, although FORMAT, DATA, and ENTRY statements may appear among the executable statements. Placing DATA statements among executable constructs is now an obsolescent feature. USE statements, if any, must appear first. Internal or module subprograms, if any, must appear last following a CONTAINS statement.

In Table 2-1 a vertical line separates statements and constructs that can be interspersed; a horizontal line separates statements that must not be interspersed.

Table 2-1 Requirements on statement ordering

PROGRAM, FUNCTION, SUBROUTINE, MODULE, or BLOCK DATA statement		
USE statements		
IMPORT statements ³		
FORMAT ⁵ and ENTRY ⁴ statements	IMPLICIT NONE	
	PARAMETER statements	IMPLICIT statements
	PARAMETER and DATA statements ⁶	Derived-type definitions, interface blocks, ⁷ type declaration statements, enumeration statements, procedure statements, statement function statements, ^{2,5} and specification statements
	DATA statements ¹	Executable constructs ⁵
CONTAINS statement ⁸		
Internal subprograms or module subprograms		
END statement		
1. Placing DATA statements among executable constructs is obsolescent. 2. Statement function statements are obsolescent. 3. Can appear only in interface bodies. 4. Can appear only in modules and external procedures. 5. Cannot appear in module specification parts, interface bodies, and block data subprograms. 6. Cannot appear in interface bodies. 7. Cannot appear in block data subprograms. 8. Cannot appear in internal subprograms, interface bodies, and block data subprograms.		

2.7 Example Fortran Program

Illustrated below is a very simple Fortran program consisting of one program unit, the main program. Three data objects are declared: H, T, and U. These become the loop indices in a triply-nested loop construct (8.7) containing a logical IF statement (8.4.2) that conditionally executes an input/output statement (9.4).


```
program sum_of_cubes
! This program prints all 3-digit numbers that
! equal the sum of the cubes of their digits.
implicit none
integer :: H, T, U
do H = 1, 9
    do T = 0, 9
        do U = 0, 9
            if (100*H + 10*T + U == H**3 + T**3 + U**3) &
                print "(3I1)", H, T, U
        end do
    end do
end do
end program sum_of_cubes
```

This Fortran program is standard conforming and should be compilable and executable on any standard Fortran computing system, producing the following output:

```
153
370
371
```