Section 12: Procedures

The concept of a procedure was introduced in 2.2.3. This section contains a complete description of procedures. The actions specified by a procedure are performed when the procedure is invoked by execution of a reference to it. The reference may identify, as actual arguments, entities that are associated during execution of the procedure reference with corresponding dummy arguments in the procedure definition.

12.1 Procedure classifications

A procedure is classified according to the form of its reference and the way it is defined.

12.1.1 Procedure classification by reference

The definition of a procedure specifies it to be a function or a subroutine. A reference to a function either appears explicitly as a primary within an expression, or is implied by a defined operation (7.1.3) within an expression. A reference to a subroutine is a CALL statement or a defined assignment statement (7.5.1.3).

A procedure is classified as **elemental** if it is a procedure that may be referenced elementally (12.7).

12.1.2 Procedure classification by means of definition

A procedure is either an intrinsic procedure, an external procedure, a module procedure, an internal procedure, a dummy procedure, or a statement function.

12.1.2.1 Intrinsic procedures

A procedure that is provided as an inherent part of the processor is an **intrinsic procedure**.

12.1.2.2 External, internal, and module procedures

An **external procedure** is a procedure that is defined by an external subprogram or by a means other than Fortran.

An **internal procedure** is a procedure that is defined by an internal subprogram. Internal subprograms may appear in the main program, in an external subprogram, or in a module subprogram. Internal subprograms shall not appear in other internal subprograms. Internal subprograms are the same as external subprograms except that the name of the internal procedure is not a global entity, an internal subprogram shall not contain an ENTRY statement, the internal procedure name shall not be argument associated with a dummy procedure (12.4.1.2), and the internal subprogram has access to host entities by host association.

A **module procedure** is a procedure that is defined by a module subprogram.

If a subprogram contains one or more ENTRY statements, it defines a procedure for each ENTRY statement and a procedure for the SUBROUTINE or FUNCTION statement.

12.1.2.3 Dummy procedures

A dummy argument that is specified as a procedure or appears in a procedure reference is a **dummy procedure**.

12.1.2.4 Statement functions

A function that is defined by a single statement is a **statement function** (12.5.4).

12.2 Characteristics of procedures

The **characteristics of a procedure** are the classification of the procedure as a function or subroutine, whether or not it is pure, whether or not it is elemental, the characteristics of its dummy arguments, and the characteristics of its result value if it is a function.

12.2.1 Characteristics of dummy arguments

Each dummy argument is either a dummy data object, a dummy procedure, or an asterisk (alternate return indicator). A dummy argument other than an asterisk may be specified to have the OPTIONAL attribute. This attribute means that the dummy argument need not be associated with an actual argument for any particular reference to the procedure.

12.2.1.1 Characteristics of dummy data objects

The characteristics of a dummy data object are its type, its type parameters (if any), its shape, its intent (5.1.2.3, 5.2.1), whether it is optional (5.1.2.6, 5.2.2), and whether it is a pointer (5.1.2.7, 5.2.7) or a target (5.1.2.8, 5.2.8). If a type parameter of an object or a bound of an array is an expression that depends on the value or attributes of another object, the exact dependence on other entities is a characteristic. If a shape, size, or character length is assumed, it is a characteristic.

12.2.1.2 Characteristics of dummy procedures

The characteristics of a dummy procedure are the explicitness of its interface (12.3.1), its characteristics as a procedure if the interface is explicit, and whether it is optional (5.1.2.6, 5.2.2).

12.2.1.3 Characteristics of asterisk dummy arguments

An asterisk as a dummy argument has no characteristics.

12.2.2 Characteristics of function results

The characteristics of a function result are its type, type parameters (if any), rank, and whether it is a pointer. If a function result is an array that is not a pointer, its shape is a characteristic. If a type parameter of a function result or a bound of a function result array is not a constant expression, the exact dependence on the entities in the expression is a characteristic. If the length of a character function result is assumed, this is a characteristic.

12.3 Procedure interface

The **interface** of a procedure determines the forms of reference through which it may be invoked. The interface consists of the characteristics of the procedure, the name of the procedure, the name and characteristics of each dummy argument, and the procedure's generic identifiers, if any. The characteristics of a procedure are fixed, but the remainder of the interface may differ in different scoping units.

NOTE 12.1

For more explanatory information on procedure interfaces, see section C.9.3.

12.3.1 Implicit and explicit interfaces

If a procedure is accessible in a scoping unit, its interface is either **explicit** or **implicit** in that scoping unit. The interface of an internal procedure, module procedure, or intrinsic procedure is always explicit in such a scoping unit. The interface of a recursive subroutine or a recursive function with a separate result name is explicit within the subprogram that defines it. The interface of a statement function is always implicit. The interface of an external procedure or dummy procedure is explicit in a scoping unit other than its own if an interface block (12.3.2.1) for the procedure is supplied or accessible, and implicit otherwise.

NOTE 12.2

> For example, the subroutine LLS of C.8.3.5 has an explicit interface.

12.3.1.1 Explicit interface

A procedure other than a statement function shall have an explicit interface if

(1) A reference to the procedure appears
  (a) With an argument keyword (12.4.1),
  (b) As a reference by its generic name (12.3.2.1),
  (c) As a defined assignment (subroutines only),
  (d) In an expression as a defined operator (functions only), or
  (e) In a context that requires it to be pure,
(2) The procedure has
  (a) An optional dummy argument,
  (b) A dummy argument that is an assumed-shape array, a pointer, or a target,
  (c) An array-valued result (functions only),
  (d) A result that is a pointer (functions only), or
  (e) A result whose character length parameter value is not assumed and not constant (character functions only), or
(3) The procedure is elemental.

12.3.2 Specification of the procedure interface

The interface for an internal, external, module, or dummy procedure is specified by a FUNCTION, SUBROUTINE, or ENTRY statement and by specification statements for the dummy arguments and the result of a function. These statements may appear in the procedure definition, in an interface block, or in both except that the ENTRY statement shall not appear in an interface block.

NOTE 12.3

> An interface block cannot be used to describe the interface of an internal procedure, a module procedure, or an intrinsic procedure because the interfaces of such procedures are already explicit. However, the name of a module procedure may appear in a MODULE PROCEDURE statement in an interface block (12.3.2.1).

12.3.2.1 Procedure interface block

R1201 *interface-block* **is** *interface-stmt*
        [*interface-specification*] ...
        end-interface-stmt

R1202 *interface-specification* **is** *interface-body*
        or *module-procedure-stmt*

R1203 *interface-stmt* **is** INTERFACE [*generic-spec*]

R1204 *end-interface-stmt* **is** END INTERFACE [*generic-spec*]

R1205 *interface-body* **is** *function-stmt*
        [*specification-part*]

end-function-stmt
or *subroutine-stmt*
[*specification-part*]
end-subroutine-stmt

Constraint: An *interface-body* of a pure procedure shall specify the intents of all dummy arguments except pointer, alternate return, and procedure arguments.

R1206 *module-procedure-stmt* **is** MODULE PROCEDURE *procedure-name-list*

R1207 *generic-spec* **is** *generic-name*
or OPERATOR (*defined-operator*)
or ASSIGNMENT (=)

Constraint: An *interface-body* shall not contain an *entry-stmt*, *data-stmt*, *format-stmt*, or *stmt-function-stmt*.

Constraint: The MODULE PROCEDURE statement is allowed only if the *interface-block* has a *generic-spec* and is in a scoping unit where each *procedure-name* is accessible as a module procedure.

Constraint: An *interface-block* in a subprogram shall not contain an *interface-body* for a procedure defined by that subprogram.

Constraint: The *generic-spec* may be included in the *end-interface-stmt* only if it was provided in the *interface-stmt* and, if included, shall be identical to the *generic-spec* in the *interface-stmt*.

Constraint: A *procedure-name* in a *module-procedure-stmt* shall not be one which previously had been specified in any *module-procedure-stmt* with the same generic identifier in the same specification part.

An external or module subprogram specifies a **specific interface** for the procedures defined in that subprogram. Such a specific interface is explicit for module procedures and implicit for external procedures. An **interface body** in an interface block specifies an explicit specific interface for an existing external procedure or a dummy procedure. If the name in a procedure heading in an interface block is the same as the name of a dummy argument in the subprogram containing the interface block, the interface block declares that dummy argument to be a dummy procedure with the indicated interface; otherwise, the interface block declares the name to be the name of an external procedure with the indicated procedure interface.

An interface body specifies all of the procedure's characteristics and these shall be consistent with those specified in the procedure definition, except that the interface may specify a procedure that is not pure if the procedure is defined to be pure. The specification part of an interface body may specify attributes or define values for data entities that do not determine characteristics of the procedure. Such specifications have no effect. An interface block shall not contain an ENTRY statement, but an entry interface may be specified by using the entry name as the procedure name in the interface body. A procedure shall not have more than one explicit specific interface in a given scoping unit.

NOTE 12.4

The dummy argument names may be different because the name of a dummy argument is not a characteristic.

NOTE 12.5

An example of an interface block without a generic specification is:

```
INTERFACE

   SUBROUTINE EXT1 (X, Y, Z)
      REAL, DIMENSION (100, 100) :: X, Y, Z
   END SUBROUTINE EXT1
```

NOTE 12.5 *(Continued)*

```
   SUBROUTINE EXT2 (X, Z)
      REAL X
      COMPLEX (KIND = 4) Z (2000)
   END SUBROUTINE EXT2

   FUNCTION EXT3 (P, Q)
      LOGICAL EXT3
      INTEGER P (1000)
      LOGICAL Q (1000)
   END FUNCTION EXT3

END INTERFACE
```

This interface block specifies explicit interfaces for the three external procedures EXT1, EXT2, and EXT3. Invocations of these procedures may use argument keywords (12.4.1); for example:

```
EXT3 (Q = P_MASK (N+1 : N+1000), P = ACTUAL_P)
```

An interface block with a generic specification specifies a **generic interface** for each of the procedures in the interface block. If the interface block is in a module or is in a scoping unit that accesses a module by use association, the MODULE PROCEDURE statement lists those module procedures that have this generic interface. The listed module procedures may be defined in the module containing the interface block or may be accessible via a USE statement. The characteristics of module procedures are not given in interface blocks, but are assumed from the module subprograms. A generic interface is always explicit.

Any procedure may be referenced via its specific interface. It also may be referenced via its generic interface, if it has one. The generic name, defined operator, or equals symbol in a generic specification is a **generic identifier** for all the procedures in the interface block. The rules on how any two procedures with the same generic identifier shall differ are given in 14.1.2.3. They ensure that any generic invocation applies to at most one specific procedure.

A **generic name** specifies a single name to reference all of the procedure names in the interface block. A generic name may be the same as any one of the procedure names in the interface block, or the same as any accessible generic name.

NOTE 12.6

An example of a generic procedure interface is:

```
INTERFACE SWITCH

   SUBROUTINE INT_SWITCH (X, Y)
      INTEGER, INTENT (INOUT) :: X, Y
   END SUBROUTINE INT_SWITCH

   SUBROUTINE REAL_SWITCH (X, Y)
      REAL, INTENT (INOUT) :: X, Y
   END SUBROUTINE REAL_SWITCH

   SUBROUTINE COMPLEX_SWITCH (X, Y)
      COMPLEX, INTENT (INOUT) :: X, Y
   END SUBROUTINE COMPLEX_SWITCH

END INTERFACE SWITCH
```

Any of these three subroutines (INT_SWITCH, REAL_SWITCH, COMPLEX_SWITCH) may be referenced with the generic name SWITCH, as well as by its specific name. For example, a reference to INT_SWITCH could take the form:

```
CALL SWITCH (MAX_VAL, LOC_VAL) ! MAX_VAL and LOC_VAL are of type INTEGER
```

12.3.2.1.1 Defined operations

If OPERATOR is specified in a generic specification, all of the procedures specified in the interface block shall be functions that may be referenced as defined operations (12.4). In the case of functions of two arguments, infix binary operator notation is implied. In the case of functions of one argument, prefix operator notation is implied. OPERATOR shall not be specified for functions with no arguments or for functions with more than two arguments. The dummy arguments shall be nonoptional dummy data objects and shall be specified with INTENT (IN) and the function result shall not have assumed character length. If the operator is an *intrinsic-operator* (R310), the number of function arguments shall be consistent with the intrinsic uses of that operator.

A defined operation is treated as a reference to the function. For a unary defined operation, the operand corresponds to the function's dummy argument; for a binary operation, the left-hand operand corresponds to the first dummy argument of the function and the right-hand operand corresponds to the second dummy argument.

NOTE 12.7

An example of the use of the OPERATOR generic specification is:

```
INTERFACE OPERATOR ( * )

   FUNCTION BOOLEAN_AND (B1, B2)
      LOGICAL, INTENT (IN) :: B1 (:), B2 (SIZE (B1))
      LOGICAL :: BOOLEAN_AND (SIZE (B1))
   END FUNCTION BOOLEAN_AND

END INTERFACE OPERATOR ( * )
```

This allows, for example

```
SENSOR (1:N) * ACTION (1:N)
```

as an alternative to the function call

```
BOOLEAN_AND (SENSOR (1:N), ACTION (1:N))    ! SENSOR and ACTION are
                                            ! of type LOGICAL
```

A given defined operator may, as with generic names, apply to more than one function, in which case it is generic in exact analogy to generic procedure names. For intrinsic operator symbols, the generic properties include the intrinsic operations they represent. Because both forms of each relational operator have the same interpretation (7.3), extending one form (such as <=) has the effect of defining both forms (<= and .LE.).

12.3.2.1.2 Defined assignments

If ASSIGNMENT is specified in an INTERFACE statement, all the procedures in the interface block shall be subroutines that may be referenced as defined assignments (7.5.1.3). Each of these subroutines shall have exactly two dummy arguments. Each argument shall be nonoptional. The first argument shall have INTENT (OUT) or INTENT (INOUT) and the second argument shall have INTENT (IN). A defined assignment is treated as a reference to the subroutine, with the left-hand side as the first argument and the right-hand side enclosed in parentheses as the second argument. The ASSIGNMENT generic specification specifies that the assignment operation is extended, or redefined if both sides of the equals sign are of the same derived type.

NOTE 12.8

An example of the use of the ASSIGNMENT generic specification is:

```
INTERFACE ASSIGNMENT ( = )

   SUBROUTINE LOGICAL_TO_NUMERIC (N, B)
      INTEGER, INTENT (OUT) :: N
```

NOTE 12.8 *(Continued)*

```
      LOGICAL, INTENT (IN)  :: B
   END SUBROUTINE LOGICAL_TO_NUMERIC

   SUBROUTINE CHAR_TO_STRING (S, C)
      USE STRING_MODULE       ! Contains definition of type STRING
      TYPE (STRING), INTENT (OUT) :: S  ! A variable-length string
      CHARACTER (*), INTENT (IN) ::  C
   END SUBROUTINE CHAR_TO_STRING

 END INTERFACE ASSIGNMENT ( = )
```

Example assignments are:

```
KOUNT = SENSOR (J)     ! CALL LOGICAL_TO_NUMERIC (KOUNT, (SENSOR (J)))
NOTE  = '89AB'         ! CALL CHAR_TO_STRING (NOTE, ('89AB'))
```

12.3.2.2 EXTERNAL statement

An **EXTERNAL statement** specifies the EXTERNAL attribute for a list of names.

R1208 *external-stmt* **is** EXTERNAL [::] *external-name-list*

Each *external-name* shall be the name of an external procedure, a dummy argument, or a block data program unit.

The appearance of the name of a dummy argument in an EXTERNAL statement specifies that the dummy argument is a dummy procedure. The appearance in an EXTERNAL statement of a name that is not the name of a dummy argument specifies that the name is the name of an external procedure or block data program unit.

If an external procedure name or a dummy procedure name is used as an actual argument, its interface shall be explicit or it shall be explicitly declared to have the EXTERNAL attribute. Appearance of an intrinsic procedure name in an EXTERNAL statement causes that name to become the name of an external procedure and thus the intrinsic procedure of the same name is not available in the scoping unit.

The appearance of the name of a block data program unit in an EXTERNAL statement confirms that the block data program unit is a part of the program.

Only one appearance of a name in all of the EXTERNAL statements in a scoping unit is permitted. A name that appears in an EXTERNAL statement in a given scoping unit or is a use-associated entity with the EXTERNAL attribute shall not also appear as a specific procedure name in an interface block in the scoping unit nor in an interface block that is accessible to the scoping unit.

NOTE 12.9

For explanatory information on potential portability problems with external procedures, see section C.9.1.

NOTE 12.10

An example of an EXTERNAL statement is:

```
   EXTERNAL FOCUS
```

12.3.2.3 INTRINSIC statement

An **INTRINSIC statement** specifies a list of names that have the INTRINSIC attribute. A name that has the INTRINSIC attribute represents an intrinsic procedure (Section 13). The INTRINSIC attribute permits a name that represents a specific intrinsic function to be used as an actual argument.

R1209 *intrinsic-stmt* **is** INTRINSIC [::] *intrinsic-procedure-name-list*

Constraint: Each *intrinsic-procedure-name* shall be the name of an intrinsic procedure.

The appearance of a name in an INTRINSIC statement confirms that the name is the name of an intrinsic procedure. The appearance of a generic intrinsic procedure name (13.11, 13.12) in an INTRINSIC statement does not cause that name to lose its generic property. In a scoping unit, a name may appear both as the name of a generic intrinsic procedure in an INTRINSIC statement and as the name of a generic interface, provided that the procedures in the interface and the specific intrinsic procedures are all functions or all subroutines (14.1.2.3).

If the specific name of an intrinsic function (13.13) is used as an actual argument, the name shall either appear in an INTRINSIC statement or be given the INTRINSIC attribute in a type declaration statement in the scoping unit.

Only one appearance of a name in all of the INTRINSIC statements in a scoping unit is permitted.

NOTE 12.11

A name shall not appear in both an EXTERNAL and an INTRINSIC statement in the same scoping unit.

12.3.2.4 Implicit interface specification

In a scoping unit where the interface of a function is implicit, the type and type parameters of the function result are specified by an implicit or explicit type specification of the function name. The type, type parameters, and shape of dummy arguments of a procedure referenced from a scoping unit where the interface of the procedure is implicit shall be such that the actual arguments are consistent with the characteristics of the dummy arguments.

12.4 Procedure reference

The form of a procedure reference is dependent on the interface of the procedure, but is independent of the means by which the procedure is defined. The forms of procedure references are:

R1210 *function-reference* **is** *function-name* ([*actual-arg-spec-list*])

Constraint: The *actual-arg-spec-list* for a function reference shall not contain an *alt-return-spec*.

R1211 *call-stmt* **is** CALL *subroutine-name* [([*actual-arg-spec-list*])]

A function may also be referenced as a defined operation. A subroutine may also be referenced as a defined assignment.

R1212 *actual-arg-spec* **is** [*keyword* =] *actual-arg*

R1213 *keyword* **is** *dummy-arg-name*

R1214 *actual-arg* **is** *expr*
or *variable*
or *procedure-name*
or *alt-return-spec*

R1215 *alt-return-spec* **is** * *label*

Constraint: The *keyword* = shall not appear if the interface of the procedure is implicit in the scoping unit.

Constraint: The *keyword* = may be omitted from an *actual-arg-spec* only if the *keyword* = has been omitted from each preceding *actual-arg-spec* in the argument list.

Constraint: Each *keyword* shall be the name of a dummy argument in the explicit interface of the procedure.

Constraint: A non-intrinsic elemental procedure shall not be used as an actual argument.

Constraint: A *procedure-name actual-arg* shall not be the name of an internal procedure or of a statement function and shall not be the generic name of a procedure unless it is also a specific name (12.3.2.1, 13.1).

NOTE 12.12

This standard does not allow internal procedures to be used as actual arguments, in part to simplify the problem of ensuring that internal procedures with recursive hosts access entities from the correct instance of the host. If, as an extension, a processor allows internal procedures to be used as actual arguments, the correct instance in this case is the instance in which the procedure is supplied as an actual argument, even if the corresponding dummy argument is eventually invoked from a different instance.

Constraint: In a reference to a pure procedure, a *procedure-name actual-arg* shall be the name of a pure procedure (12.6).

NOTE 12.13

This constraint ensures that the purity of a procedure cannot be undermined by allowing it to call a nonpure procedure.

Constraint: The *label* used in the *alt-return-spec* shall be the statement label of a branch target statement that appears in the same scoping unit as the *call-stmt*.

NOTE 12.14

Successive commas shall not be used to omit optional arguments.

12.4.1 Actual arguments, dummy arguments, and argument association

In either a subroutine reference or a function reference, the actual argument list identifies the correspondence between the actual arguments supplied and the dummy arguments of the procedure. In the absence of an argument keyword, an actual argument is associated with the dummy argument occupying the corresponding position in the dummy argument list; that is, the first actual argument is associated with the first dummy argument, the second actual argument is associated with the second dummy argument, etc. If an argument keyword is present, the actual argument is associated with the dummy argument whose name is the same as the argument keyword (using the dummy argument names from the interface accessible in the scoping unit containing the procedure reference). Exactly one actual argument shall be associated with each nonoptional dummy argument. At most one actual argument may be associated with each optional dummy argument. Each actual argument shall be associated with a dummy argument.

NOTE 12.15

For example, the procedure defined by

```
SUBROUTINE SOLVE (FUNCT, SOLUTION, METHOD, STRATEGY, PRINT)
   INTERFACE
      FUNCTION FUNCT (X)
         REAL FUNCT, X
      END FUNCTION FUNCT
   END INTERFACE
   REAL SOLUTION
   INTEGER, OPTIONAL :: METHOD, STRATEGY, PRINT
   ...
```

may be invoked with

```
CALL SOLVE (FUN, SOL, PRINT = 6)
```

provided its interface is explicit; if the interface is specified by an interface block, the name of the last argument shall be PRINT.

12.4.1.1 guments associated with dummy data objects

If a dummy argument is a dummy data object, the associated actual argument shall be an expression of the same type or a data object of the same type. The kind type parameter value of the actual argument shall agree with that of the dummy argument. The value of the character length parameter of an actual argument of type nondefault character shall agree with that of the dummy argument. If the dummy argument is an assumed-shape array of type default character, the value of the character length parameter of the actual argument shall agree with that of the dummy argument. If the dummy argument is an assumed-shape array, the rank of the dummy argument shall agree with the rank of the actual argument.

If a scalar dummy argument is of type default character, the length *len* of the dummy argument shall be less than or equal to the length of the actual argument. The dummy argument becomes associated with the leftmost *len* characters of the actual argument. If an array dummy argument is of type default character, the restriction on length is for the entire array and not for each array element. The length of an array element in the dummy argument array may be different from the length of an array element in the associated actual argument array, array element, or array element substring, but the dummy argument array shall not extend beyond the end of the actual argument array.

Except when a procedure reference is elemental (12.7), each element of an array-valued actual argument or of a sequence in a sequence association (12.4.1.4) is associated with the element of the dummy array that has the same position in array element order (6.2.2.2).

NOTE 12.16

For type default character sequence associations, the interpretation of element is provided in 12.4.1.4.

If the dummy argument is a pointer, the actual argument shall be a pointer and the types, type parameters, and ranks shall agree.

At the invocation of the procedure, the dummy argument pointer receives the pointer association status of the actual argument. If the actual argument is currently associated, the dummy argument becomes associated with the same target. The association status may change during the execution of the procedure. When execution of the procedure completes, the pointer association status of the dummy argument becomes undefined if it is associated with a target that becomes undefined (14.7.6); following this, the pointer association status of the actual argument becomes that of the dummy argument.

If the dummy argument is not a pointer and the corresponding actual argument is a pointer, the actual argument shall be currently associated with a target and the dummy argument becomes argument associated with that target.

If the dummy argument does not have the TARGET or POINTER attribute, any pointers associated with the actual argument do not become associated with the corresponding dummy argument on invocation of the procedure. If such a dummy argument is associated with a dummy argument with the TARGET attribute, whether any pointers associated with the original actual argument become associated with the dummy argument with the TARGET attribute is processor dependent.

If the dummy argument has the TARGET attribute and is either a scalar or an assumed-shape array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript

(1) Any pointers associated with the actual argument become associated with the corresponding dummy argument on invocation of the procedure and

(2) When execution of the procedure completes, any pointers associated with the dummy argument remain associated with the actual argument.

If the dummy argument has the TARGET attribute and is an explicit-shape array or is an assumed-size array, and the corresponding actual argument has the TARGET attribute but is not an array section with a vector subscript

(1) On invocation of the procedure, whether any pointers associated with the actual argument become associated with the corresponding dummy argument is processor dependent and

(2) When execution of the procedure completes, the pointer association status of any pointer that is pointer associated with the dummy argument is processor dependent.

If the dummy argument has the TARGET attribute and the corresponding actual argument does not have the TARGET attribute or is an array section with a vector subscript, any pointers associated with the dummy argument become undefined when execution of the procedure completes.

If the actual argument is scalar, the corresponding dummy argument shall be scalar unless the actual argument is an element of an array that is not an assumed-shape or pointer array, or a substring of such an element. If the procedure is nonelemental and is referenced by a generic name or as a defined operator or defined assignment, the ranks of the actual arguments and corresponding dummy arguments shall agree.

If a dummy argument is an assumed-shape array, the actual argument shall not be an assumed-size array or a scalar (including an array element designator or an array element substring designator).

A scalar dummy argument of a nonelemental procedure may be associated only with a scalar actual argument.

If a dummy argument has INTENT (OUT) or INTENT (INOUT), the actual argument shall be definable. If a dummy argument has INTENT (OUT), the corresponding actual argument becomes undefined at the time the association is established.

If the actual argument is an array section having a vector subscript, the dummy argument is not definable and shall not have INTENT (OUT) or INTENT (INOUT).

NOTE 12.17

Argument intent specifications serve several purposes in addition to documenting the intended use of dummy arguments. A processor can check whether an INTENT (IN) dummy argument is used in a way that could redefine it. A slightly more sophisticated processor could check to see whether an INTENT (OUT) dummy argument could possibly be referenced before it is defined. If the procedure's interface is explicit, the processor can also verify that actual arguments corresponding to INTENT (OUT) or INTENT (INOUT) dummy arguments are definable. A more sophisticated processor could use this information to optimize the translation of the referencing scoping unit by taking advantage of the fact that actual arguments corresponding to INTENT (IN) dummy arguments will not be changed and that any prior value of an actual argument corresponding to an INTENT (OUT) dummy argument will not be referenced and could thus be discarded.

INTENT (OUT) means that the value of the argument after invoking the procedure is entirely the result of executing that procedure. If there is any possibility that an argument should retain its current value rather than being redefined, INTENT (INOUT) should be used rather than INTENT (OUT), even if there is no explicit reference to the value of the dummy argument. Because an INTENT(OUT) variable is considered undefined on entry to the procedure, any default initialization specified for its type will be applied.

INTENT (INOUT) is not equivalent to omitting the INTENT attribute. The argument corresponding to an INTENT (INOUT) dummy argument always shall be definable, while an argument corresponding to a dummy argument without an INTENT attribute need be definable only if the dummy argument is actually redefined.

NOTE 12.18

For more explanatory information on argument association and evaluation, see section C.9.4. For more explanatory information on pointers and targets as dummy arguments, see section C.9.5.

12.4.1.2 Actual arguments associated with dummy procedures

If a dummy argument is a dummy procedure, the associated actual argument shall be the specific name of an external, module, dummy, or intrinsic procedure. The only intrinsic procedures permitted are those listed in 13.13 and not marked with a bullet (•). If the specific name is also a generic name, only the specific procedure is associated with the dummy argument.

If the interface of the dummy procedure is explicit, the characteristics listed in 12.2 shall be the same for the associated actual procedure and the corresponding dummy procedure, except that a pure actual procedure may be associated with a dummy procedure that is not pure and an elemental intrinsic actual procedure may be associated with a dummy procedure that is not elemental.

If the interface of the dummy procedure is implicit and either the name of the dummy procedure is explicitly typed or the procedure is referenced as a function, the dummy procedure shall not be referenced as a subroutine and the actual argument shall be a function or dummy procedure.

If the interface of the dummy procedure is implicit and a reference to the procedure appears as a subroutine reference, the actual argument shall be a subroutine or dummy procedure.

12.4.1.3 Actual arguments associated with alternate return indicators

If a dummy argument is an asterisk (12.5.2.3), the associated actual argument shall be an alternate return specifier. The label in the alternate return specifier shall identify an executable construct in the scoping unit containing the procedure reference.

12.4.1.4 Sequence association

An actual argument represents an **element sequence** if it is an array expression, an array element designator, or an array element substring designator. If the actual argument is an array expression, the element sequence consists of the elements in array element order. If the actual argument is an array element designator, the element sequence consists of that array element and each element that follows it in array element order.

If the actual argument is of type default character and is an array expression, array element, or array element substring designator, the element sequence consists of the character storage units beginning with the first storage unit of the actual argument and continuing to the end of the array. The character storage units of an array element substring designator are viewed as array elements consisting of consecutive groups of character storage units having the character length of the dummy array.

NOTE 12.19

Some of the elements in the element sequence may consist of storage units from different elements of the original array.

An actual argument that represents an element sequence and corresponds to a dummy argument that is an array-valued data object is sequence associated with the dummy argument if the dummy argument is an explicit-shape or assumed-size array. The rank and shape of the actual argument need not agree with the rank and shape of the dummy argument, but the number of elements in the dummy argument shall not exceed the number of elements in the element sequence of the actual argument. If the dummy argument is assumed-size, the number of elements in the dummy argument is exactly the number of elements in the element sequence.

12.4.1.5 Restrictions on dummy arguments not present

A dummy argument is **present** in an instance of a subprogram if it is associated with an actual argument and the actual argument either is a dummy argument that is present in the invoking subprogram or is not a dummy argument of the invoking subprogram. A dummy argument that is not optional shall be present. An optional dummy argument that is not present is subject to the following restrictions:

(1) If it is a dummy data object, it shall not be referenced or be defined. If it is of a type for which default initialization is specified for some component, the initialization has no effect.

(2) If it is a dummy procedure, it shall not be invoked.

(3) It shall not be supplied as an actual argument corresponding to a nonoptional dummy argument other than as the argument of the PRESENT intrinsic function.

(4) A subobject of it shall not be supplied as an actual argument corresponding to an optional dummy argument.

(5) If it is an array, it shall not be supplied as an actual argument to an elemental procedure unless an array of the same rank is supplied as an actual argument corresponding to a nonoptional dummy argument of that elemental procedure.

(6) If it is a pointer, it shall not be supplied as an actual argument corresponding to a nonpointer dummy argument other than as the argument of the PRESENT intrinsic function.

Except as noted in the list above, it may be supplied as an actual argument corresponding to an optional dummy argument, which is then also considered not to be associated with an actual argument.

12.4.1.6 Restrictions on entities associated with dummy arguments

While an entity is associated with a dummy argument, the following restrictions hold:

(1) No action that affects the allocation status of the entity may be taken. Action that affects the value of the entity or any part of it shall be taken through the dummy argument unless

  (a) the dummy argument has the POINTER attribute,

  (b) the part is all or part of a pointer subobject, or

  (c) the dummy argument has the TARGET attribute, the dummy argument does not have INTENT (IN), the dummy argument is a scalar object or an assumed-shape array, and the actual argument is a target other than an array section with a vector subscript.

NOTE 12.20

In

```
SUBROUTINE OUTER
   REAL, POINTER :: A (:)
   ...
   ALLOCATE (A (1:N))
   ...
   CALL INNER (A)
   ...
CONTAINS
   SUBROUTINE INNER (B)
      REAL :: B (:)
      ...

   END SUBROUTINE INNER
   SUBROUTINE SET (C, D)
      REAL, INTENT (OUT) :: C
      REAL, INTENT (IN) :: D
      C = D
   END SUBROUTINE SET
END SUBROUTINE OUTER
```

NOTE 12.20 *(Continued)*

an assignment statement such as

```
A (1) = 1.0
```

would not be permitted during the execution of INNER because this would be changing A without using B, but statements such as

```
B (1) = 1.0
```

or

```
CALL SET (B (1), 1.0)
```

would be allowed. Similarly,

```
DEALLOCATE (A)
```

would not be allowed because this affects the allocation of B without using B. In this case,

```
DEALLOCATE (B)
```

also would not be permitted If B were declared with the POINTER attribute, either of the statements

```
DEALLOCATE (A)
```

and

```
DEALLOCATE (B)
```

would be permitted, but not both.

NOTE 12.21

If there is a partial or complete overlap between the actual arguments associated with two different dummy arguments of the same procedure and the dummy arguments have neither the POINTER nor TARGET attribute, the overlapped portions shall not be defined, redefined, or become undefined during the execution of the procedure. For example, in

```
CALL SUB (A (1:5), A (3:9))
```

A (3:5) shall not be defined, redefined, or become undefined through the first dummy argument because it is part of the argument associated with the second dummy argument and shall not be defined, redefined, or become undefined through the second dummy argument because it is part of the argument associated with the first dummy argument. A (1:2) remains definable through the first dummy argument and A (6:9) remains definable through the second dummy argument.

NOTE 12.22

This restriction applies equally to pointer targets. In

```
REAL, DIMENSION (10), TARGET :: A
REAL, DIMENSION (:), POINTER :: B, C
B => A (1:5)
C => A (3:9)
CALL SUB (B, C)  ! The dummy arguments of SUB are neither pointers nor targets.
```

B (3:5) cannot be defined because it is part of the argument associated with the second dummy argument. C (1:3) cannot be defined because it is part of the argument associated with the first dummy argument. A (1:2) [which is B (1:2)] remains definable through the first dummy argument and A (6:9) [which is C (4:7)] remains definable through the second dummy argument.

NOTE 12.23

Since a dummy argument declared with an intent of IN shall not be used to change the associated actual argument, the associated actual argument remains constant throughout the execution of the procedure.

(2) If the value of any part of the entity is affected through the dummy argument, then at any time during the execution of the procedure, either before or after the definition, it may be referenced only through that dummy argument unless

  (a) the dummy argument has the POINTER attribute,

  (b) the part is all or part of a pointer subobject, or

  (c) the dummy argument has the TARGET attribute, the dummy argument does not have INTENT (IN), the dummy argument is a scalar object or an assumed-shape array, and the actual argument is a target other than an array section with a vector subscript.

NOTE 12.24

In

```
MODULE DATA
   REAL :: W, X, Y, Z
END MODULE DATA

PROGRAM MAIN
   USE DATA
      ...
   CALL INIT (X)
      ...
END PROGRAM MAIN

SUBROUTINE INIT (V)
   USE DATA
      ...
   READ (*, *) V
      ...
END SUBROUTINE INIT
```

variable X shall not be directly referenced at any time during the execution of INIT because it is being defined through the dummy argument V. X may be (indirectly) referenced through V. W, Y, and Z may be directly referenced. X may, of course, be directly referenced once execution of INIT is complete.

NOTE 12.25

The restrictions on entities associated with dummy arguments are intended to facilitate a variety of optimizations in the translation of the subprogram, including implementations of argument association in which the value of an actual argument that is neither a pointer nor a target is maintained in a register or in local storage.

12.4.2 Function reference

A function is invoked during expression evaluation by a *function-reference* or by a defined operation (7.1.3). When it is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the function is executed. When execution of the function is complete, the value of the function result is available for use in the expression that caused the function to be invoked. The characteristics of the function result (12.2.2) are determined by the interface of the function. A reference to an elemental function (12.7) is an elemental reference if one or more actual arguments are arrays and all array arguments have the same shape.

12.4.3 Subroutine reference

A subroutine is invoked by execution of a CALL statement or defined assignment statement (7.5.1.3). When a subroutine is invoked, all actual argument expressions are evaluated, then the arguments are associated, and then the subroutine is executed. When the actions specified by the subroutine are completed, execution of the CALL statement or defined assignment statement is also completed. If a CALL statement includes one or more alternate return specifiers among its arguments, control may be transferred to one of the statements indicated, depending on the action specified by the subroutine. A reference to an elemental subroutine (12.7) is an elemental reference if all actual arguments corresponding to INTENT (OUT) and INTENT (INOUT) dummy arguments are arrays that have the same shape and the remaining actual arguments are conformable with them.

12.5 Procedure definition

12.5.1 Intrinsic procedure definition

Intrinsic procedures are defined as an inherent part of the processor. A standard-conforming processor shall include the intrinsic procedures described in Section 13, but may include others. However, a standard-conforming program shall not make use of intrinsic procedures other than those described in Section 13.

12.5.2 Procedures defined by subprograms

When a procedure defined by a subprogram is invoked, an instance (12.5.2.4) of the subprogram is created and executed. Execution begins with the first executable construct following the FUNCTION, SUBROUTINE, or ENTRY statement specifying the name of the procedure invoked or with the END statement if there is no other executable construct.

12.5.2.1 Effects of INTENT attribute on subprograms

The INTENT attribute of dummy data objects limits the way in which they may be used in a subprogram. A dummy data object having INTENT (IN) shall neither be defined nor become undefined during the execution of the procedure. A dummy data object having INTENT (OUT) is initially undefined in the subprogram unless the object is of a type for which default initialization is specified. A dummy data object with INTENT (INOUT) may be referenced or be defined. A dummy data object whose intent is not specified is subject to the limitations of the data entity that is the associated actual argument. That is, a reference to the dummy data object may occur if the actual argument is defined and the dummy data object may be defined if the actual argument is definable.

12.5.2.2 Function subprogram

A **function subprogram** is a subprogram that has a FUNCTION statement as its first statement.

R1216 *function-subprogram* **is** *function-stmt*
[*specification-part*]
[*execution-part*]
[*internal-subprogram-part*]
end-function-stmt

R1217 *function-stmt* **is** [*prefix*] FUNCTION *function-name* ■
■ ([*dummy-arg-name-list*]) [RESULT (*result-name*)]

Constraint: If RESULT is specified, the *function-name* shall not appear in any specification statement in the scoping unit of the function subprogram.

R1218 *prefix* **is** *prefix-spec* [*prefix-spec*] ...

R1219 *prefix-spec* **is** *type-spec*
or RECURSIVE
or PURE
or ELEMENTAL

Constraint: A *prefix* shall contain at most one of each *prefix-spec*.

Constraint: If ELEMENTAL is present, RECURSIVE shall not be present.

R1220 *end-function-stmt* **is** END [FUNCTION [*function-name*]]

Constraint: If RESULT is specified, *result-name* shall not be the same as *function-name*.

Constraint: FUNCTION shall be present in the *end-function-stmt* of an internal or module function.

Constraint: An internal function subprogram shall not contain an ENTRY statement.

Constraint: An internal function subprogram shall not contain an *internal-subprogram-part*.

Constraint: If a *function-name* is present in the *end-function-stmt*, it shall be identical to the *function-name* specified in the *function-stmt*.

The type and type parameters (if any) of the result of the function defined by a function subprogram may be specified by a type specification in the FUNCTION statement or by the name of the result variable appearing in a type statement in the declaration part of the function subprogram. It shall not be specified both ways. If it is not specified either way, it is determined by the implicit typing rules in force within the function subprogram. If the function result is array-valued or a pointer, this shall be specified by specifications of the name of the result variable within the function body. The specifications of the function result attributes, the specification of dummy argument attributes, and the information in the procedure heading collectively define the characteristics of the function (12.2).

The *prefix-spec* RECURSIVE shall be present if the function directly or indirectly invokes itself or a function defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall be present if a function defined by an ENTRY statement in the subprogram directly or indirectly invokes itself, another function defined by an ENTRY statement in that subprogram, or the function defined by the FUNCTION statement.

The name of the function is *function-name*.

If RESULT is specified, the name of the result variable of the function is *result-name*, its characteristics (12.2.2) are those of the function result, and all occurrences of the function name in *execution-part* statements in the scoping unit are recursive function references. If RESULT is not specified, the result variable is *function-name* and all occurrences of the function name in *execution-part* statements in the scoping unit are references to the result variable. The value of the result variable at the completion of execution of the function is the value returned by the function. If the function result has been declared to be a pointer, the shape of the value returned by the function is determined by the shape of the result variable when the execution of the function is completed. If the result variable is not a pointer, its value shall be defined by the function. If the function result has been declared a pointer, the function shall either associate a target with the result variable pointer or cause the association status of this pointer to become defined as disassociated.

NOTE 12.26

The result variable is similar to any other variable local to a function subprogram. Its existence begins when execution of the function is initiated and ends when execution of the function is terminated. However, because the final value of this variable is used subsequently in the evaluation of the expression that invoked the function, an implementation may wish to defer releasing the storage occupied by that variable until after its value has been used in expression evaluation.

If the *prefix-spec* PURE or ELEMENTAL is present, the subprogram is a pure subprogram and shall meet the additional constraints of 12.6.

If the *prefix-spec* ELEMENTAL is present, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.7.1.

If both RECURSIVE and RESULT are specified, the interface of the function being defined is explicit within the function subprogram.

NOTE 12.27

An example of a recursive function is:

```
RECURSIVE FUNCTION CUMM_SUM (ARRAY) RESULT (C_SUM)
   REAL, INTENT (IN), DIMENSION (:) :: ARRAY
   REAL, DIMENSION (SIZE (ARRAY)) ::C_SUM
   INTEGER N
   N = SIZE (ARRAY)
   IF (N .LE. 1) THEN
      C_SUM = ARRAY
   ELSE
      N = N / 2
      C_SUM (:N) = CUMM_SUM (ARRAY (:N))
      C_SUM (N+1:) = C_SUM (N) + CUMM_SUM (ARRAY (N+1:))
   END IF
END FUNCTION CUMM_SUM
```

12.5.2.3 Subroutine subprogram

A **subroutine subprogram** is a subprogram that has a SUBROUTINE statement as its first statement.

R1221 *subroutine-subprogram* **is** *subroutine-stmt*
    [*specification-part*]
    [*execution-part*]
    [*internal-subprogram-part*]
    end-subroutine-stmt

R1222 *subroutine-stmt* **is** [*prefix*] SUBROUTINE *subroutine-name* ■
    ■ [([*dummy-arg-list*])]

Constraint: The *prefix* of a *subroutine-stmt* shall not contain a *type-spec*.

R1223 *dummy-arg* **is** *dummy-arg-name*
    or *

R1224 *end-subroutine-stmt* **is** END [SUBROUTINE [*subroutine-name*]]

Constraint: SUBROUTINE shall be present in the *end-subroutine-stmt* of an internal or module subroutine.

Constraint: An internal subroutine subprogram shall not contain an ENTRY statement.

Constraint: An internal subroutine subprogram shall not contain an *internal-subprogram-part*.

Constraint: If a *subroutine-name* is present in the *end-subroutine-stmt*, it shall be identical to the *subroutine-name* specified in the *subroutine-stmt*.

The *prefix-spec* RECURSIVE shall be present if the subroutine directly or indirectly invokes itself or a subroutine defined by an ENTRY statement in the same subprogram. Similarly, RECURSIVE shall be present if a subroutine defined by an ENTRY statement in the subprogram directly or indirectly invokes itself, another subroutine defined by an ENTRY statement in that subprogram, or the subroutine defined by the SUBROUTINE statement.

If RECURSIVE is specified, the interface of the subroutine being defined is explicit within the subroutine subprogram.

The name of the subroutine is *subroutine-name*.

If the *prefix-spec* PURE or ELEMENTAL is present, the subprogram is a pure subprogram and shall meet the additional constraints of 12.6.

If the *prefix-spec* ELEMENTAL is present, the subprogram is an elemental subprogram and shall meet the additional constraints of 12.7.1.

12.5.2.4 Instances of a subprogram

When a function or subroutine defined by a subprogram is invoked, an **instance** of that subprogram is created. When a statement function is invoked, an instance of that statement function is created.

Each instance has an independent sequence of execution and an independent set of dummy arguments and local nonsaved data objects. If an internal procedure or statement function in the subprogram is invoked directly from an instance of the subprogram or from an internal subprogram or statement function that has access to the entities of that instance, the created instance of the internal subprogram or statement function also has access to the entities of that instance of the host subprogram.

All other entities are shared by all instances of the subprogram.

NOTE 12.28

The value of a saved data object appearing in one instance may have been defined in a previous instance or by initialization in a DATA statement or type declaration statement.

12.5.2.5 ENTRY statement

An **ENTRY statement** permits a procedure reference to begin with a particular executable statement within the function or subroutine subprogram in which the ENTRY statement appears.

R1225 *entry-stmt* **is** ENTRY *entry-name* [([*dummy-arg-list*]) ■
■ [RESULT (*result-name*)]]

Constraint: If RESULT is specified, the *entry-name* shall not appear in any specification or type-declaration statement in the scoping unit of the function program.

Constraint: An *entry-stmt* may appear only in an *external-subprogram* or *module-subprogram*. An *entry-stmt* shall not appear within an *executable-construct*.

Constraint: RESULT may be present only if the *entry-stmt* is in a function subprogram.

Constraint: Within the subprogram containing the *entry-stmt*, the *entry-name* shall not appear as a dummy argument in the FUNCTION or SUBROUTINE statement or in another ENTRY statement and it shall not appear in an EXTERNAL or INTRINSIC statement.

Constraint: A *dummy-arg* may be an alternate return indicator only if the ENTRY statement is in a subroutine subprogram.

Constraint: If RESULT is specified, *result-name* shall not be the same as *entry-name*.

Optionally, a subprogram may have one or more ENTRY statements.

If the ENTRY statement is in a function subprogram, an additional function is defined by that subprogram. The name of the function is *entry-name* and its result variable is *result-name* or is *entry-name* if no *result-name* is provided. The characteristics of the function result are specified by specifications of the result variable. The dummy arguments of the function are those specified in the ENTRY statement. If the characteristics of the result of the function named in the ENTRY statement are the same as the characteristics of the result of the function named in the FUNCTION statement, their result variables identify the same variable, although their names need not be the same. Otherwise, they are storage associated and shall all be scalars without the POINTER attribute and one of the types: default integer, default real, double precision real, default complex, or default logical.

If RESULT is specified in the ENTRY statement and RECURSIVE is specified in the FUNCTION statement, the interface of the function defined by the ENTRY statement is explicit within the function subprogram.

If the ENTRY statement is in a subroutine subprogram, an additional subroutine is defined by that subprogram. The name of the subroutine is *entry-name*. The dummy arguments of the subroutine are those specified in the ENTRY statement.

If RECURSIVE is specified in the SUBROUTINE statement, the interface of the subroutine defined by the ENTRY statement is explicit within the subroutine subprogram.

The order, number, types, kind type parameters, and names of the dummy arguments in an ENTRY statement may differ from the order, number, types, kind type parameters, and names of the dummy arguments in the FUNCTION or SUBROUTINE statement in the containing program.

Because an ENTRY statement defines an additional function or an additional subroutine, it is referenced in the same manner as any other function or subroutine (12.4).

In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear in an executable statement preceding that ENTRY statement, unless it also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.

In a subprogram, a name that appears as a dummy argument in an ENTRY statement shall not appear in the expression of a statement function unless the name is also a dummy argument of the statement function, appears in a FUNCTION or SUBROUTINE statement, or appears in an ENTRY statement that precedes the statement function statement.

If a dummy argument appears in an executable statement, the execution of the executable statement is permitted during the execution of a reference to the function or subroutine only if the dummy argument appears in the dummy argument list of the procedure name referenced.

If a dummy argument is used in a specification expression to specify an array bound or character length of an object, the appearance of the object in a statement that is executed during a procedure reference is permitted only if the dummy argument appears in the dummy argument list of the procedure name referenced and it is present (12.4.1.5).

A scoping unit containing a reference to a procedure defined by an ENTRY statement may have access to an interface body for the procedure. The procedure header for the interface body shall be a FUNCTION statement for an entry in a function subprogram and shall be a SUBROUTINE statement for an entry in a subroutine subprogram.

The keyword RECURSIVE is not used in an ENTRY statement. Instead, the presence or absence of RECURSIVE in the initial SUBROUTINE or FUNCTION statement controls whether the procedure defined by an ENTRY statement is permitted to reference itself.

The keyword PURE is not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is pure if and only if PURE or ELEMENTAL is specified in the SUBROUTINE or FUNCTION statement.

The keyword ELEMENTAL is not used in an ENTRY statement. Instead, the procedure defined by an ENTRY statement is elemental if and only if ELEMENTAL is specified in the SUBROUTINE or FUNCTION statement.

12.5.2.6 RETURN statement

R1226 *return-stmt* **is** RETURN [*scalar-int-expr*]

Constraint: The *return-stmt* shall be in the scoping unit of a function or subroutine subprogram.

Constraint: The *scalar-int-expr* is allowed only in the scoping unit of a subroutine subprogram.

Execution of the **RETURN statement** completes execution of the instance of the subprogram in which it appears. If the expression is present and has a value *n* between 1 and the number of asterisks in the dummy argument list, the CALL statement that invoked the subroutine transfers control to the statement identified by the *n*th alternate return specifier in the actual argument list. If the expression is omitted or has a value outside the required range, there is no transfer of control to an alternate return.

Execution of an *end-function-stmt* or *end-subroutine-stmt* is equivalent to executing a RETURN statement with no expression.

12.5.2.7 CONTAINS statement

R1227 *contains-stmt* **is** CONTAINS

The **CONTAINS statement** separates the body of a main program, module, or subprogram from any internal or module subprograms it may contain. The CONTAINS statement is not executable.

12.5.3 Definition of procedures by means other than Fortran

The means other than Fortran by which a procedure may be defined are processor dependent. A reference to such a procedure is made as though it were defined by an external subprogram. The definition of a non-Fortran procedure shall not be in a Fortran program unit and a Fortran program unit shall not be in the definition of a non-Fortran procedure. The interface to a non-Fortran procedure may be specified in an interface block.

NOTE 12.29

For explanatory information on definition of procedures by means other than Fortran, see section C.9.2.

12.5.4 Statement function

A statement function is a function defined by a single statement.

R1228 *stmt-function-stmt* **is** *function-name* ([*dummy-arg-name-list*]) = *scalar-expr*

Constraint: The *primaries* of the *scalar-expr* shall be constants (literal and named), references to variables, references to functions and function dummy procedures, and intrinsic operations. If *scalar-expr* contains a reference to a function or a function dummy procedure, the reference shall not require an explicit interface, the function shall not require an explicit interface unless it is an intrinsic, the function shall not be a transformational intrinsic, and the result shall be scalar. If an argument to a function or a function dummy procedure is array valued, it shall be an array name. If a reference to a statement function appears in *scalar-expr*, its definition shall have been provided earlier in the scoping unit and shall not be the name of the statement function being defined.

Constraint: Named constants in *scalar-expr* shall have been declared earlier in the scoping unit or made accessible by use or host association. If array elements appear in *scalar-expr*, the parent array shall have been declared as an array earlier in the scoping unit or made accessible by use or host association.

Constraint: If a *dummy-arg-name*, variable, function reference, or dummy function reference is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm this implied type and the values of any implied type parameters.

Constraint: The *function-name* and each *dummy-arg-name* shall be specified, explicitly or implicitly, to be scalar.

Constraint: A given *dummy-arg-name* may appear only once in any *dummy-arg-name-list*.

Constraint: Each variable reference in *scalar-expr* may be either a reference to a dummy argument of the statement function or a reference to a variable accessible in the same scoping unit as the statement function statement.

The definition of a statement function with the same name as an accessible entity from the host shall be preceded by the declaration of its type in a type declaration statement.

The dummy arguments have a scope of the statement function statement. Each dummy argument has the same type and type parameters as the entity of the same name in the scoping unit containing the statement function.

A statement function shall not be supplied as a procedure argument.

The value of a statement function reference is obtained by evaluating the expression using the values of the actual arguments for the values of the corresponding dummy arguments and, if necessary, converting the result to the declared type and type attributes of the function.

A function reference in the scalar expression shall not cause a dummy argument of the statement function to become redefined or undefined.

12.6 Pure procedures

A pure procedure is

(1) A pure intrinsic function (13.1),
(2) A pure intrinsic subroutine (13.10),
(3) Defined by a pure subprogram, or
(4) A statement function that references only pure functions.

A pure subprogram is a subprogram that has the *prefix-spec* PURE or ELEMENTAL. The following additional constraints apply to the syntax rules defining non-intrinsic pure function subprograms (R1216-R1220) or non-intrinsic pure subroutine subprograms (R1221-R1224).

Constraint: The *specification-part* of a pure function subprogram shall specify that all dummy arguments have INTENT (IN) except procedure arguments and arguments with the POINTER attribute.

Constraint: The *specification-part* of a pure subroutine subprogram shall specify the intents of all dummy arguments except procedure arguments, alternate return indicators, and arguments with the POINTER attribute.

Constraint: A local variable declared in the *specification-part* or *internal-subprogram-part* of a pure subprogram shall not have the SAVE attribute.

NOTE 12.30

Variable initialization in a *type-declaration-stmt* or a *data-stmt* implies the SAVE attribute; therefore, such initialization is also disallowed.

Constraint: The *specification-part* of a pure subprogram shall specify that all dummy arguments that are procedure arguments are pure.

Constraint: If a procedure that is neither an intrinsic procedure nor a statement function is used in a context that requires it to be pure, then its interface shall be explicit in the scope of that use. The interface shall specify that the procedure is pure.

NOTE 12.31

It is expected that most mathematical library procedures will be pure. This form of restriction allows these procedures to be used in contexts where they are not required to be pure without the need for an *interface-block*.

Constraint: All internal subprograms in a pure subprogram shall be pure.

Constraint: In a pure subprogram any variable which is in common or accessed by host or use association, is a dummy argument to a pure function, is a dummy argument with INTENT (IN) to a pure subroutine, or an object that is storage associated with any such variable, shall not be used in the following contexts:

(1) As the *variable* of an *assignment-stmt*;
(2) As a DO variable or implied DO variable;
(3) As an *input-item* in a *read-stmt* from an internal file;
(4) As an *internal-file-unit* in a *write-stmt*;
(5) As an IOSTAT= specifier in an input or output statement with an internal file;
(6) As the *pointer-object* of a *pointer-assignment-stmt*;
(7) As the *tar* of a *pointer-assignment-stmt*;
(8) As the *expr* of an *assignment-stmt* in which the *variable* is of a derived type if the derived type has a pointer component at any level of component selection;

NOTE 12.32

This requires that processors be able to determine if entities with the PRIVATE attribute or with private components have a pointer component.

(9) As an *allocate-object* or *stat-variable* in an *allocate-stmt* or *deallocate-stmt*, or as a *pointer-object* in a *nullify-stmt*; or

(10) As an actual argument associated with a dummy argument with INTENT (OUT) or INTENT (INOUT) or with the POINTER attribute.

Constraint: Any procedure referenced in a pure subprogram, including one referenced via a defined operation or assignment, shall be pure.

Constraint: A pure subprogram shall not contain a *print-stmt*, *open-stmt*, *close-stmt*, *backspace-stmt*, *endfile-stmt*, *rewind-stmt*, or *inquire-stmt*.

Constraint: A pure subprogram shall not contain a *read-stmt* or *write-stmt* whose *io-unit* is an *external-file-unit* or *.

Constraint: A pure subprogram shall not contain a *stop-stmt*.

NOTE 12.33

The above constraints are designed to guarantee that a pure procedure is free from side effects (i.e., modifications of data visible outside the procedure), which means that it is safe to reference it in constructs such as a FORALL *assignment-stmt* where there is no explicit order of evaluation.

The constraints on pure subprograms may appear complicated, but it is not necessary for a programmer to be intimately familiar with them. From the programmer's point of view, these constraints can be summarized as follows: a pure subprogram shall not contain any operation that could conceivably result in an assignment or pointer assignment to a common variable, a variable accessed by use or host association, or an INTENT (IN) dummy argument; nor shall a pure subprogram contain any operation that could conceivably perform any external file I/O or STOP operation. Note the use of the word conceivably; it is not sufficient for a pure subprogram merely to be side-effect free in practice. For example, a function that contains an assignment to a global variable but in a block that is not executed in any invocation of the function is nevertheless not a pure function. The exclusion of functions of this nature is required if strict compile-time checking is to be used.

It is expected that most library procedures will conform to the constraints required of pure procedures, and so can be declared pure and referenced in FORALL statements and constructs and within user-defined pure procedures. It is also anticipated that most library procedures will not reference global data. Referencing global data may inhibit concurrent execution.

NOTE 12.34

Pure subroutines are included to allow subroutine calls from pure procedures in a safe way, and to allow *forall-assignment*s to be defined assignments. The constraints for pure subroutines are based on the same principles as for pure functions, except that side effects to INTENT (OUT), INTENT (INOUT), and pointer dummy arguments are permitted.

12.7 Elemental procedures

12.7.1 Elemental procedure declaration and interface

An elemental procedure is an elemental intrinsic procedure or a procedure that is defined by an elemental subprogram.

An elemental subprogram has the *prefix-spec* ELEMENTAL. An elemental subprogram is a pure subprogram. The PURE *prefix-spec* need not be present; it is implied by the ELEMENTAL *prefix-spec*. The following additional constraints apply to the syntax rules defining elemental function subprograms (R1216-R1220) or elemental subroutine subprograms (R1221-R1224).

Constraint: All dummy arguments shall be scalar and shall not have the POINTER attribute.

Constraint: For a function, the result shall be scalar and shall not have the POINTER attribute.

Constraint: A dummy argument, or a subobject thereof, shall not appear in a *specification-expr* except as the argument to one of the intrinsic functions BIT_SIZE, KIND, LEN, or the numeric inquiry functions (13.11.8).

Constraint: A *dummy-arg* shall not be *.

Constraint: A *dummy-arg* shall not be a dummy procedure.

NOTE 12.35

An elemental subprogram is a pure subprogram and all of the constraints for pure subprograms also apply.

Note 12.36

The restriction on dummy arguments in specification expressions is imposed primarily to facilitate optimization. An example of usage that is not permitted is

```
ELEMENTAL REAL FUNCTION F (A, N)
   REAL, INTENT (IN) :: A
   INTEGER, INTENT (IN) :: N
   REAL :: WORK_ARRAY(N)    ! Invalid
   ...
END FUNCTION F
```

An example of usage that is permitted is

```
ELEMENTAL REAL FUNCTION F (A)
   REAL, INTENT (IN) :: A
   REAL (SELECTED_REAL_KIND (PRECISION (A)*2)) :: WORK
   ...
END FUNCTION F
```

12.7.2 Elemental function actual arguments and results

If a generic name or a specific name is used to reference an elemental function, the shape of the result is the same as the shape of the actual argument with the greatest rank. If the actual arguments are all scalar, the result is scalar. For those elemental functions that have more than one argument, all actual arguments shall be conformable. In the array-valued case, the values of the elements, if any, of the result are the same as would have been obtained if the scalar-valued function had been applied separately, in any order, to corresponding elements of each array actual argument.

NOTE 12.37

An example of an elemental reference to the intrinsic function MAX:

if X and Y are arrays of shape (M, N),

```
MAX (X, 0.0, Y)
```

is an array expression of shape (M, N) whose elements have values

```
MAX (X(I, J),  0.0, Y(I, J)), I = 1, 2, ..., M, J = 1,2, ..., N
```

12.7.3 Elemental subroutine actual arguments

An elemental subroutine is one that has only scalar dummy arguments, but may have array actual arguments. In a reference to an elemental subroutine, either all actual arguments shall be scalar, or all actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy arguments shall be arrays of the same shape and the remaining actual arguments shall be conformable with them. In the case that the actual arguments associated with INTENT (OUT) and INTENT (INOUT) dummy arguments are arrays, the values of the elements, if any, of the results are the same as would be obtained if the subroutine had been applied separately, in any order, to corresponding elements of each array actual argument.

In a reference to the intrinsic subroutine MVBITS, the actual arguments corresponding to the TO and FROM dummy arguments may be the same variable. Apart from this, the actual arguments in a reference to an elemental subroutine must satisfy the restrictions of 12.4.1.6.