Section 7: Expressions and assignment

This section describes the formation, interpretation, and evaluation rules for expressions and the assignment statement.

7.1 Expressions

An **expression** represents either a data reference or a computation, and its value is either a scalar or an array. An expression is formed from operands, operators, and parentheses.

NOTE 7.1

Simple forms of an operand are constants and variables, such as:

```
3.0

.FALSE.

A

B (I)

C (I:J)
```

An operand is either a scalar or an array. An operation is either intrinsic (7.2) or defined (7.3). More complicated expressions can be formed using operands which are themselves expressions.

NOTE 7.2

Examples of intrinsic operators are:

```
+

*

>

.AND.
```

7.1.1 Form of an expression

Evaluation of an expression produces a value, which has a type, type parameters (if appropriate), and a shape (7.1.4).

NOTE 7.3

Examples of expressions are:

```
A + B

(A - B) * C

A ** B

C .AND. D

F // G
```

An expression is defined in terms of several categories: primary, level-1 expression, level-2 expression, level-3 expression, level-4 expression, and level-5 expression.

These categories are related to the different operator precedence levels and, in general, are defined in terms of other categories. The simplest form of each expression category is a *primary*. The rules given below specify the syntax of an expression. The semantics are specified in 7.2 and 7.3.

7.1.1.1 Primary

R701 *primary* **is** *constant*
or *constant-subobject*
or *variable*
or *array-constructor*
or *structure-constructor*
or *function-reference*
or (*expr*)

R702 *constant-subobject* **is** *subobject*

Constraint: *subobject* shall be a subobject designator whose parent is a constant.

A variable that is a primary shall not be a whole assumed-size array.

NOTE 7.4

Examples of a *primary* are:

Example	Syntactic class
`1.0`	*constant*
`'ABCDEFGHIJKLMNOPQRSTUVWXYZ' (I:I)`	*constant-subobject*
`A`	*variable*
`(/ 1.0, 2.0 /)`	*array-constructor*
`PERSON (12, 'Jones')`	*structure-constructor*
`F (X, Y)`	*function-reference*
`(S + T)`	(*expr*)

7.1.1.2 Level-1 expressions

Defined unary operators have the highest operator precedence (Table 7.8). Level-1 expressions are primaries optionally operated on by defined unary operators:

R703 *level-1-expr* **is** [*defined-unary-op*] *primary*

R704 *defined-unary-op* **is** **.** *letter* [*letter*] ... **.**

Constraint: A *defined-unary-op* shall not contain more than 31 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.5

Simple examples of a level-1 expression are:

Example	Syntactic class
`A`	*primary* (R701)
`.INVERSE. B`	*level-1-expr* (R703)

A more complicated example of a level-1 expression is:

`.INVERSE. (A + B)`

7.1.1.3 Level-2 expressions

Level-2 expressions are level-1 expressions optionally involving the numeric operators *power-op*, *mult-op*, and *add-op*.

R705 *mult-operand* **is** *level-1-expr* [*power-op mult-operand*]

R706 *add-operand* **is** [*add-operand mult-op*] *mult-operand*

R707 *level-2-expr* **is** [[*level-2-expr*] *add-op*] *add-operand*

R708 *power-op* **is** **

R709 *mult-op* **is** *
or /

R710 *add-op* **is** +
or –

NOTE 7.6

Simple examples of a level-2 expression are:

Example	Syntactic class	Remarks
`A`	*level-1-expr*	A is a *primary*. (R703)
`B ** C`	*mult-operand*	B is a *level-1-expr*, ** is a *power-op*, and C is a *mult-operand*. (R705)
`D * E`	*add-operand*	D is an *add-operand*, * is a *mult-op*, and E is a *mult-operand*. (R706)
`+1`	*level-2-expr*	+ is an *add-op* and 1 is an *add-operand*. (R707)
`F – I`	*level-2-expr*	F is a *level-2-expr*, – is an *add-op*, and I is an *add-operand*. (R707)

A more complicated example of a level-2 expression is:

```
– A + D * E + B ** C
```

7.1.1.4 Level-3 expressions

Level-3 expressions are level-2 expressions optionally involving the character operator *concat-op*.

R711 *level-3-expr* **is** [*level-3-expr concat-op*] *level-2-expr*

R712 *concat-op* **is** //

NOTE 7.7

Simple examples of a level-3 expression are:

Example	Syntactic class
`A`	*level-2-expr* (R707)
`B // C`	*level-3-expr* (R711)

A more complicated example of a level-3 expression is:

```
X // Y // 'ABCD'
```

7.1.1.5 Level-4 expressions

Level-4 expressions are level-3 expressions optionally involving the relational operators *rel-op*.

R713 *level-4-expr* **is** [*level-3-expr rel-op*] *level-3-expr*

R714 *rel-op* **is** .EQ.
or .NE.
or .LT.
or .LE.
or .GT.
or .GE.
or ==
or /=
or <
or <=
or >

or >=

NOTE 7.8

Simple examples of a level-4 expression are:

Example	Syntactic class
`A`	*level-3-expr* (R711)
`B .EQ. C`	*level-4-expr* (R713)
`D < E`	*level-4-expr* (R713)

A more complicated example of a level-4 expression is:

`(A + B) .NE. C`

7.1.1.6 Level-5 expressions

Level-5 expressions are level-4 expressions optionally involving the logical operators *not-op*, *and-op*, *or-op*, and *equiv-op*.

R715 *and-operand* **is** [*not-op*] *level-4-expr*

R716 *or-operand* **is** [*or-operand and-op*] *and-operand*

R717 *equiv-operand* **is** [*equiv-operand or-op*] *or-operand*

R718 *level-5-expr* **is** [*level-5-expr equiv-op*] *equiv-operand*

R719 *not-op* **is** .NOT.

R720 *and-op* **is** .AND.

R721 *or-op* **is** .OR.

R722 *equiv-op* **is** .EQV.
or .NEQV.

NOTE 7.9

Simple examples of a level-5 expression are:

Example	Syntactic class
`A`	*level-4-expr* (R713)
`.NOT. B`	*and-operand* (R715)
`C .AND. D`	*or-operand* (R716)
`E .OR. F`	*equiv-operand* (R717)
`G .EQV. H`	*level-5-expr* (R718)
`S .NEQV. T`	*level-5-expr* (R718)

A more complicated example of a level-5 expression is:

`A .AND. B .EQV. .NOT. C`

7.1.1.7 General form of an expression

Expressions are level-5 expressions optionally involving defined binary operators. Defined binary operators have the lowest operator precedence (Table 7.8).

R723 *expr* **is** [*expr defined-binary-op*] *level-5-expr*

R724 *defined-binary-op* **is** . *letter* [*letter*] ... .

Constraint: A *defined-binary-op* shall not contain more than 31 letters and shall not be the same as any *intrinsic-operator* or *logical-literal-constant*.

NOTE 7.10

Simple examples of an expression are:

Example	Syntactic class
`A`	*level-5-expr* (R718)
`B.UNION.C`	*expr* (R723)

More complicated examples of an expression are:

```
(B .INTERSECT. C) .UNION. (X - Y)

A + B .EQ. C * D

.INVERSE. (A + B)

A + B .AND. C * D

E // G .EQ. H (1:10)
```

7.1.2 Intrinsic operations

An **intrinsic operation** is either an intrinsic unary operation or an intrinsic binary operation. An **intrinsic unary operation** is an operation of the form *intrinsic-operator* x_2 where x_2 is of an intrinsic type (4.3) listed in Table 7.1 for the unary intrinsic operator.

An **intrinsic binary operation** is an operation of the form x_1 *intrinsic-operator* x_2 where x_1 and x_2 are of the intrinsic types (4.3) listed in Table 7.1 for the binary intrinsic operator and are in shape conformance (7.1.5).

Table 7.1 Type of operands and results for intrinsic operators

Intrinsic operator *op*	Type of x_1	Type of x_2	Type of $[x_1]$ *op* x_2
Unary +, –		I, R, Z	I, R, Z
Binary +, –, *, /, **	I	I, R, Z	I, R, Z
	R	I, R, Z	R, R, Z
	Z	I, R, Z	Z, Z, Z
//	C	C	C
.EQ., .NE., ==, /=	I	I, R, Z	L, L, L
	R	I, R, Z	L, L, L
	Z	I, R, Z	L, L, L
	C	C	L
.GT., .GE., .LT., .LE. >, >=, <, <=	I	I, R	L, L
	R	I, R	L, L
	C	C	L
.NOT.		L	L
.AND., .OR., .EQV., .NEQV.	L	L	L

Note: The symbols I, R, Z, C, and L stand for the types integer, real, complex, character, and logical, respectively. Where more than one type for x_2 is given, the type of the result of the operation is given in the same relative position in the next column. For the intrinsic operators requiring operands of type character, the kind type parameters of the operands shall be the same.

A **numeric intrinsic operation** is an intrinsic operation for which the *intrinsic-operator* is a numeric operator (+, –, *, /, or **). A **numeric intrinsic operator** is the operator in a numeric intrinsic operation.

For numeric intrinsic binary operations, the two operands may be of different numeric types or different kind type parameters. Except for a value raised to an integer power, if the operands have different types or kind type parameters, the effect is as if each operand that differs in type or kind type parameter from those of the result is converted to the type and kind type parameter of the result before the operation is performed. When a value of type real or complex is raised to an integer power, the integer operand need not be converted.

A **character intrinsic operation**, **relational intrinsic operation**, and **logical intrinsic operation** are similarly defined in terms of a **character intrinsic operator** (//), **relational intrinsic operator** (.EQ., .NE., .GT., .GE., .LT., .LE., ==, /=, >, >=, <, and <=), and **logical intrinsic operator** (.AND., .OR., .NOT., .EQV., and .NEQV.), respectively. For the character intrinsic operator //, the kind type parameters shall be the same.

A **numeric relational intrinsic operation** is a relational intrinsic operation where the operands are of numeric type. A **character relational intrinsic operation** is a relational intrinsic operation where the operands are of type character and have the same kind type parameter value.

7.1.3 Defined operations

A **defined operation** is either a defined unary operation or a defined binary operation. A **defined unary operation** is an operation that has the form *defined-unary-op* x_2 and that is defined by a function and a generic interface block (12.3.2.1) or that has the form *intrinsic-operator* x_2 where the type of x_2 is not that required for the unary intrinsic operation (7.1.2), and that is defined by a function and a generic interface block.

A **defined binary operation** is an operation that has the form x_1 *defined-binary-op* x_2 and that is defined by a function and a generic interface block or that has the form x_1 *intrinsic-operator* x_2 where the types or ranks of either x_1 or x_2 or both are not those required for the intrinsic binary operation (7.1.2), and that is defined by a function and a generic interface block.

NOTE 7.11

An intrinsic operator may be used as the operator in a defined operation. In such a case, the generic properties of the operator are extended.

An **extension operation** is a defined operation in which the operator is of the form *defined-unary-op* or *defined-binary-op*. Such an operator is called an **extension operator**. The operator used in an extension operation may be such that a generic interface for the operator may specify more than one function.

A **defined elemental operation** is a defined operation for which the function is elemental (12.7).

7.1.4 Data type, type parameters, and shape of an expression

The data type and shape of an expression depend on the operators and on the data types and shapes of the primaries used in the expression, and are determined recursively from the syntactic form of the expression. The data type of an expression is one of the intrinsic types (4.3) or a derived type (4.4).

R725 *logical-expr* **is** *expr*

Constraint: *logical-expr* shall be of type logical.

R726 *char-expr* **is** *expr*

Constraint: *char-expr* shall be of type character.

R727 *default-char-expr* **is** *expr*

Constraint: *default-char-expr* shall be of type default character.

R728 *int-expr* **is** *expr*

Constraint: *int-expr* shall be of type integer.

R729 *numeric-expr* **is** *expr*

Constraint: *numeric-expr* shall be of type integer, real or complex.

An expression whose type is intrinsic has a kind type parameter. In addition, an expression of type character has a character length parameter. The type parameters for an expression are determined from the form of the expression.

7.1.4.1 Data type, type parameters, and shape of a primary

The data type, type parameters, and shape of a primary are determined according to whether the primary is a constant, variable, array constructor, structure constructor, function reference, or parenthesized expression. If a primary is a constant, its type, type parameters, and shape are those of the constant. If it is a structure constructor, it is scalar and its type is determined by the constructor name. If it is an array constructor, its type, type parameters, and shape are as described in 4.5. If it is a variable or function reference, its type, type parameters, and shape are those of the variable (5.1.1, 5.1.2) or the function reference (12.4.2), respectively. In the case of a function reference, the function may be generic (12.3.2.1, 13.11), in which case its type, type parameters, and shape are determined by the types, type parameters, and ranks of its actual arguments.

If a primary is a parenthesized expression, its type, type parameters, and shape are those of the expression.

If a pointer appears as one of the following, the associated target object is referenced:

(1) A primary in an intrinsic or defined operation,

(2) As the *expr* of a parenthesized primary, or

(3) As the only primary on the right-hand side of an intrinsic assignment statement.

The type, type parameters, and shape of the primary are those of the current target. If the pointer is not associated with a target, it may appear as a primary only as an actual argument in a reference to a procedure whose corresponding dummy argument is declared to be a pointer, or as the target in a pointer assignment statement.

The intrinsic function NULL (13.14.79) returns a disassociated pointer. A disassociated pointer has no shape but does have rank. The data type, type parameters, and rank of the result of the intrinsic function NULL when it appears without an argument are determined by the pointer that becomes associated with the result. See Table 7.2.

Table 7.2 Type, type parameters, and rank of the result of NULL ( )

Appearance of NULL ( )	Type, type parameters, and rank of result:
right side of a pointer assignment	pointer on the left side
initialization for an object in a declaration	the object
default initialization for a component	the component
in a structure constructor	the corresponding component
as an actual argument	the corresponding dummy argument
in a DATA statement	the corresponding pointer object

The optional argument of the intrinsic function NULL shall be present when a reference to the intrinsic function appears as an actual argument in a reference to a generic procedure if the type, type parameters, or rank is required to resolve the generic reference.

NOTE 7.12

For example:

```
INTERFACE GEN
    SUBROUTINE S1 (J, PI)
        INTEGER J
        INTEGER, POINTER :: PI
    END SUBROUTINE S1
    SUBROUTINE S2 (K, PR)
        INTEGER K
        REAL, POINTER :: PR
    END SUBROUTINE S2
END INTERFACE
REAL, POINTER :: REAL_PTR
CALL GEN (7, NULL (REAL_PTR) )      ! Invokes S2
```

7.1.4.2 Data type, type parameters, and shape of the result of an operation

The type of the result of an intrinsic operation $[x_1]$ *op* x_2 is specified by Table 7.1. The type of the result of a defined operation $[x_1]$ *op* x_2 is specified by the function defining the operation (7.3).

The shape of the result of an intrinsic operation is the shape of x_2 if *op* is unary or if x_1 is scalar, and is the shape of x_1 otherwise.

An expression of an intrinsic type has a kind type parameter. An expression of type character also has a character length parameter. For an expression x_1 // x_2 where x_1 and x_2 are of type character, the character length parameter is the sum of the lengths of the operands and the kind type parameter is the kind type parameter of x_1, which shall be the same as the kind type parameter of x_2. For an expression *op* x_2 where *op* is an intrinsic unary operator and x_2 is of type integer, real, complex, or logical, the kind type parameter of the expression is that of the operand. For an expression x_1 *op* x_2 where *op* is a numeric intrinsic binary operator with one operand of type integer and the other of type real or complex, the kind type parameter of the expression is that of the real or complex operand. For an expression x_1 *op* x_2 where *op* is a numeric intrinsic binary operator with both operands of the same type and kind type parameters, or with one real and one complex with the same kind type parameters, the kind type parameter of the expression is identical to that of each operand. In the case where both operands are integer with different kind type parameters, the kind type parameter of the expression is that of the operand with the greater decimal exponent range or is processor dependent if the operands have the same decimal exponent range. In the case where both operands are any of type real or complex with different kind type parameters, the kind type parameter of the expression is that of the operand with the greater decimal precision or is processor dependent if the operands have the same decimal precision. For an expression x_1 *op* x_2 where *op* is a logical intrinsic binary operator with both operands of the same kind type parameter, the kind type parameter of the expression is identical to that of each operand. In the case where both operands are of type logical with different kind type parameters, the kind type parameter of the expression is processor dependent. For an expression x_1 *op* x_2 where *op* is a relational intrinsic operator, the expression has the default logical kind type parameter.

7.1.5 Conformability rules for elemental operations

An **elemental operation** is an intrinsic operation or a defined elemental operation. Two entities are in **shape conformance** if both are arrays of the same shape, or one or both are scalars.

For all elemental binary operations, the two operands shall be in shape conformance. In the case where one is a scalar and the other an array, the scalar is treated as if it were an array of the same shape as the array operand with every element, if any, of the array equal to the value of the scalar.

7.1.6 Scalar and array expressions

An expression is either a scalar expression or an array expression.

NOTE 7.13

> The following is an example of a scalar expression:
>
> ```
> Q + 2.3 * R
> ```
>
> where Q and R are scalars.
>
> The following is an example of an array expression:
>
> ```
> A (1:10) + B (2:11)
> ```
>
> where A and B are arrays.

7.1.6.1 Constant expression

A **constant expression** is an expression in which each operation is intrinsic and each primary is

(1) A constant or subobject of a constant,

(2) An array constructor where each element and the bounds and strides of each implied-DO are expressions whose primaries are constant expressions,

(3) A structure constructor where each component is a constant expression,

(4) An elemental intrinsic function reference where each argument is a constant expression,

(5) A transformational intrinsic function reference where each argument is a constant expression,

(6) A reference to the transformational intrinsic function NULL,

(7) A reference to an intrinsic function that is

  (a) an array inquiry function (13.11.15) other than ALLOCATED,
  (b) the bit inquiry function BIT_SIZE,
  (c) the character inquiry function LEN,
  (d) the kind inquiry function KIND, or
  (e) a numeric inquiry function (13.11.8)

  and where each argument of the function is

  (a) a constant expression or
  (b) a variable whose type parameters or bounds inquired about are not
    (i) assumed,
    (ii) defined by an expression that is not a constant expression, or
    (iii) definable by an ALLOCATE or pointer assignment statement,

(8) An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are constant expressions, or

(9) A constant expression enclosed in parentheses,

and where each subscript, section subscript, substring starting point, and substring ending point is a constant expression.

A **character constant expression** is a constant expression whose type is character. An **integer constant expression** is a constant expression whose type is integer. A **logical constant expression** is a constant expression whose type is logical. A **numeric constant expression** is a constant expression whose type is integer, real, or complex.

An **initialization expression** is a constant expression in which the exponentiation operation is permitted only with an integer power, and each primary is

(1) A constant or subobject of a constant,

(2) An array constructor where each element and the bounds and strides of each implied-DO are expressions whose primaries are initialization expressions,

(3) A structure constructor where each component is an initialization expression,

(4) An elemental intrinsic function reference of type integer or character where each argument is an initialization expression of type integer or character,

(5) A reference to one of the transformational functions REPEAT, RESHAPE, SELECTED_INT_KIND, SELECTED_REAL_KIND, TRANSFER, or TRIM, where each argument is an initialization expression,

(6) A reference to the transformational intrinsic function NULL,

(7) A reference to an intrinsic function that is

    (a) an array inquiry function (13.11.15) other than ALLOCATED,
    (b) the bit inquiry function BIT_SIZE,
    (c) the character inquiry function LEN,
    (d) the kind inquiry function KIND, or
    (e) a numeric inquiry function (13.11.8)

    and where each argument of the function is

    (a) an initialization expression or
    (b) a variable whose properties inquired about are not
        (i) assumed,
        (ii) defined by an expression that is not an initialization expression, or
        (iii) definable by an ALLOCATE or pointer assignment statement,

(8) An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are initialization expressions, or

(9) An initialization expression enclosed in parentheses,

and where each subscript, section subscript, substring starting point, and substring ending point is an initialization expression.

R730 *initialization-expr* **is** *expr*

Constraint: *initialization-expr* shall be an initialization expression.

R731 *char-initialization-expr* **is** *char-expr*

Constraint: *char-initialization-expr* shall be an initialization expression.

R732 *int-initialization-expr* **is** *int-expr*

Constraint: *int-initialization-expr* shall be an initialization expression.

R733 *logical-initialization-expr* **is** *logical-expr*

Constraint: *logical-initialization-expr* shall be an initialization expression.

If an initialization expression includes a reference to an inquiry function for a type parameter or an array bound of an object specified in the same *specification-part*, the type parameter or array bound shall be specified in a prior specification of the *specification-part*. The prior specification may be to the left of the inquiry function in the same statement.

NOTE 7.14

The following are examples of constant expressions:

```
3
-3 + 4
'AB'
'AB' // 'CD'
('AB' // 'CD') // 'EF'
SIZE (A)
DIGITS (X) + 4
```

where A is an explicit-shaped array with constant bounds and X is of type default real.

The following are examples of constant expressions that are not initialization expressions:

```
ABS (9.0)                                  ! Not an integer argument
3.0 ** 2.0                                 ! Not an integer power
DOT_PRODUCT ( (/ 2, 3 /), (/ 1, 7 /) )  ! Not an allowed function
```

7.1.6.2 Specification expression

A **specification expression** is an expression with limitations that make it suitable for use in specifications such as character lengths (R510) and array bounds (R515, R516). A **constant specification expression** is a specification expression that is also a constant expression.

R734 *specification-expr* **is** *scalar-int-expr*

Constraint: The *scalar-int-expr* shall be a restricted expression.

A **restricted expression** is an expression in which each operation is intrinsic and each primary is

(1) A constant or subobject of a constant,

(2) A variable that is a dummy argument that has neither the OPTIONAL nor the INTENT (OUT) attribute, or a variable that is a subobject of such a dummy argument,

(3) A variable that is in a common block or a variable that is a subobject of a variable in a common block,

(4) A variable that is made accessible by use association or host association or a variable that is a subobject of such a variable,

(5) An array constructor where each element and the bounds and strides of each implied-DO are expressions whose primaries are restricted expressions,

(6) A structure constructor where each component is a restricted expression,

(7) A reference to an intrinsic function that is

  (a) an array inquiry function (13.11.15) other than ALLOCATED,

  (b) the bit inquiry function BIT_SIZE,

  (c) the character inquiry function LEN,

  (d) the kind inquiry function KIND, or

  (e) a numeric inquiry function (13.11.8)

  and where each argument of the function is

  (a) a restricted expression or

  (b) a variable whose properties inquired about are not

    - (i) dependent on the upper bound of the last dimension of an assumed-size array,
    - (ii) defined by an expression that is not a restricted expression, or
    - (iii) definable by an ALLOCATE or pointer assignment statement,
- (8) A reference to any other intrinsic function defined in this standard where each argument is a restricted expression,
- (9) A reference to a specification function where each argument is a restricted expression,
- (10) An implied-DO variable within an array constructor where the bounds and strides of the corresponding implied-DO are restricted expressions, or
- (11) A restricted expression enclosed in parentheses,

and where any subscript, section subscript, substring starting point, or substring ending point is a restricted expression.

A function is a **specification function** if it is a pure function, is not an intrinsic function, is not an internal function, is not a statement function, does not have a dummy procedure argument, and is not defined with the RECURSIVE keyword.

NOTE 7.15

Specification functions are nonintrinsic functions that may be used in specification expressions to determine the attributes of data objects. The requirement that they be pure ensures that they cannot have side effects that could affect other objects being declared in the same *specification-part*. The requirement that they not be internal ensures that they cannot inquire, via host association, about other objects being declared in the same *specification-part*. Some requirement against direct recursion is necessary: since specification expressions must be evaluated before the first executable statement, there would be no way to break such a recursion. Indirect recursion in specification functions appears to be possible but difficult to implement, and of little value to the user, and so there is a general prohibition against recursive specification functions.

A variable in a specification expression shall have its type and type parameters, if any, specified by a previous declaration in the same scoping unit, or by the implicit typing rules in effect for the scoping unit, or by host or use association. If a variable in a specification expression is typed by the implicit typing rules, its appearance in any subsequent type declaration statement shall confirm the implied type and type parameters.

If a specification expression includes a reference to an inquiry function for a type parameter or an array bound of an entity specified in the same *specification-part*, the type parameter or array bound shall be specified in a prior specification of the *specification-part*. The prior specification may be to the left of the inquiry function reference in the same statement. If a specification expression includes a reference to the value of an element of an array specified in the same *specification-part*, the array shall be completely specified in prior declarations.

NOTE 7.16

The following are examples of specification expressions:

```
LBOUND (B, 1) + 5  ! B is an assumed-shape dummy array

M + LEN (C)        ! M and C are dummy arguments

2 * PRECISION (A)  ! A is a real variable made accessible

                   ! by a USE statement
```

7.1.7 Evaluation of operations

This section applies to both intrinsic and defined operations.

Any variable or function reference used as an operand in an expression shall be defined at the time the reference is executed. If the operand is a pointer, it shall be associated with a target object that is defined. All of the characters in a character data object reference shall be defined.

When a reference to an array or an array section is made, all of the selected elements shall be defined. When a structure is referenced, all of the components shall be defined.

The execution of any numeric operation whose result is not defined by the arithmetic used by the processor is prohibited. Raising a negative-valued primary of type real to a real power is prohibited.

The evaluation of a function reference shall neither affect nor be affected by the evaluation of any other entity within the statement. If a function reference causes definition or undefinition of an actual argument of the function, that argument or any associated entities shall not appear elsewhere in the same statement. However, execution of a function reference in the logical expression in an IF statement (8.1.2.4), the mask expression in a WHERE statement (7.5.3.1), or the subscripts and strides in a FORALL statement (7.5.4) is permitted to define variables in the statement that is conditionally executed.

NOTE 7.17

For example, the statements

```
A (I) = F (I)
Y = G (X) + X
```

are prohibited if the reference to F defines or undefines I or the reference to G defines or undefines X.

However, in the statements

```
IF (F (X)) A = X
WHERE (G (X)) B = X
```

F or G may define X.

The type of an expression in which a function reference appears does not affect, and is not affected by, the evaluation of the actual arguments of the function.

Execution of an array element reference requires the evaluation of its subscripts. The type of an expression in which the array element reference appears does not affect, and is not affected by, the evaluation of its subscripts. Execution of an array section reference requires the evaluation of its section subscripts. The type of an expression in which an array section appears does not affect, and is not affected by, the evaluation of the array section subscripts. Execution of a substring reference requires the evaluation of its substring expressions. The type of an expression in which a substring appears does not affect, and is not affected by, the evaluation of the substring expressions. It is not necessary for a processor to evaluate any subscript expressions or substring expressions for an array of zero size or a character entity of zero length.

The appearance of an array constructor requires the evaluation of the bounds and stride of any array constructor implied-DO it may contain. The type of an expression in which an array constructor appears does not affect, and is not affected by, the evaluation of such bounds and stride expressions.

When an elemental binary operation is applied to a scalar and an array or to two arrays of the same shape, the operation is performed element-by-element on corresponding array elements of the array operands. The processor may perform the element-by-element operations in any order.

NOTE 7.18

For example, the array expression

```
A + B
```

produces an array the same shape as A and B. The individual array elements of the result have the values of the first element of A added to the first element of B, the second element of A added to the second element of B, etc.

When an elemental unary operator operates on an array operand, the operation is performed element-by-element, in any order, and the result is the same shape as the operand.

7.1.7.1 Evaluation of operands

It is not necessary for a processor to evaluate all of the operands of an expression, or to evaluate entirely each operand, if the value of the expression can be determined otherwise.

NOTE 7.19

This principle is most often applicable to logical expressions, zero-sized arrays, and zero-length strings, but it applies to all expressions.

For example, in evaluating the expression

```
X > Y .OR. L (Z)
```

where X, Y, and Z are real and L is a function of type logical, the function reference L (Z) need not be evaluated if X is greater than Y. Similarly, in the array expression

```
W (Z) + X
```

where X is of size zero and W is a function, the function reference W (Z) need not be evaluated.

If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference.

NOTE 7.20

In the preceding examples, evaluation of these expressions causes Z to become undefined if L or W defines its argument.

7.1.7.2 Integrity of parentheses

The sections that follow state certain conditions under which a processor may evaluate an expression that is different from the one specified by applying the rules given in 7.1.1, 7.2, and 7.3. However, any expression in parentheses shall be treated as a data entity.

NOTE 7.21

For example, in evaluating the expression A + (B – C) where A, B, and C are of numeric types, the difference of B and C shall be evaluated before the addition operation is performed; the processor shall not evaluate the mathematically equivalent expression (A + B) – C.

7.1.7.3 Evaluation of numeric intrinsic operations

The rules given in 7.2.1 specify the interpretation of a numeric intrinsic operation. Once the interpretation has been established in accordance with those rules, the processor may evaluate any mathematically equivalent expression, provided that the integrity of parentheses is not violated.

Two expressions of a numeric type are mathematically equivalent if, for all possible values of their primaries, their mathematical values are equal. However, mathematically equivalent expressions of numeric type may produce different computational results.

NOTE 7.22

Any difference between the values of the expressions (1./3.)*3. and 1. is a computational difference, not a mathematical difference.

The mathematical definition of integer division is given in 7.2.1.1.

NOTE 7.23

The difference between the values of the expressions 5/2 and 5./2. is a mathematical difference, not a computational difference.

NOTE 7.24

The following are examples of expressions with allowable alternative forms that may be used by the processor in the evaluation of those expressions. A, B, and C represent arbitrary real or complex operands; I and J represent arbitrary integer operands; and X, Y, and Z represent arbitrary operands of numeric type.

Expression	Allowable alternative form
`X + Y`	`Y + X`
`X * Y`	`Y * X`
`-X + Y`	`Y - X`
`X + Y + Z`	`X + (Y + Z)`
`X - Y + Z`	`X - (Y - Z)`
`X * A / Z`	`X * (A / Z)`
`X * Y - X * Z`	`X * (Y - Z)`
`A / B / C`	`A / (B * C)`
`A / 5.0`	`0.2 * A`

The following are examples of expressions with forbidden alternative forms that shall not be used by a processor in the evaluation of those expressions.

Expression	Nonallowable alternative form
`I / 2`	`0.5 * I`
`X * I / J`	`X * (I / J)`
`I / J / A`	`I / (J * A)`
`(X + Y) + Z`	`X + (Y + Z)`
`(X * Y) - (X * Z)`	`X * (Y - Z)`
`X * (Y - Z)`	`X * Y - X * Z`

In addition to the parentheses required to establish the desired interpretation, parentheses may be included to restrict the alternative forms that may be used by the processor in the actual evaluation of the expression. This is useful for controlling the magnitude and accuracy of intermediate values developed during the evaluation of an expression.

NOTE 7.25

For example, in the expression

```
A + (B - C)
```

the parenthesized expression (B – C) shall be evaluated and then added to A.

The inclusion of parentheses may change the mathematical value of an expression. For example, the two expressions

```
A * I / J
A * (I / J)
```

may have different mathematical values if I and J are of type integer.

Each operand in a numeric intrinsic operation has a data type that may depend on the order of evaluation used by the processor.

NOTE 7.26

For example, in the evaluation of the expression

```
Z + R + I
```

where Z, R, and I represent data objects of complex, real, and integer data type, respectively, the data type of the operand that is added to I may be either complex or real, depending on which pair of operands (Z and R, R and I, or Z and I) is added first.

7.1.7.4 Evaluation of the character intrinsic operation

The rules given in 7.2.2 specify the interpretation of the character intrinsic operation. A processor is only required to evaluate as much of the character intrinsic operation as is required by the context in which the expression appears.

NOTE 7.27

For example, the statements

```
CHARACTER (LEN = 2) C1, C2, C3, CF

C1 = C2 // CF (C3)
```

do not require the function CF to be evaluated, because only the value of C2 is needed to determine the value of C1 because C1 and C2 both have a length of 2.

7.1.7.5 Evaluation of relational intrinsic operations

The rules given in 7.2.3 specify the interpretation of relational intrinsic operations. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is relationally equivalent, provided that the integrity of parentheses in any expression is not violated.

NOTE 7.28

For example, the processor may choose to evaluate the expression

```
I .GT. J
```

where I and J are integer variables, as

```
J - I .LT. 0
```

Two relational intrinsic operations are relationally equivalent if their logical values are equal for all possible values of their primaries.

7.1.7.6 Evaluation of logical intrinsic operations

The rules given in 7.2.4 specify the interpretation of logical intrinsic operations. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is logically equivalent, provided that the integrity of parentheses in any expression is not violated.

NOTE 7.29

For example, for the variables L1, L2, and L3 of type logical, the processor may choose to evaluate the expression

```
L1 .AND. L2 .AND. L3
```

as

```
L1 .AND. (L2 .AND. L3)
```

Two expressions of type logical are logically equivalent if their values are equal for all possible values of their primaries.

7.1.7.7 Evaluation of a defined operation

The rules given in 7.3 specify the interpretation of a defined operation. Once the interpretation of an expression has been established in accordance with those rules, the processor may evaluate any other expression that is equivalent, provided that the integrity of parentheses is not violated.

Two expressions of derived type are equivalent if their values are equal for all possible values of their primaries.

7.2 Interpretation of intrinsic operations

The intrinsic operations are those defined in 7.1.2. These operations are divided into the following categories: numeric, character, relational, and logical. The interpretations defined in the following sections apply to both scalars and arrays; the interpretation for arrays is obtained by applying the interpretation for scalars element by element.

The type, type parameters, and interpretation of an expression that consists of an intrinsic unary or binary operation are independent of the context in which the expression appears. In particular, the type, type parameters, and interpretation of such an expression are independent of the type and type parameters of any other larger expression in which it appears.

NOTE 7.30

For example, if X is of type real, J is of type integer, and INT is the real-to-integer intrinsic conversion function, the expression INT (X + J) is an integer expression and X + J is a real expression.

7.2.1 Numeric intrinsic operations

A numeric operation is used to express a numeric computation. Evaluation of a numeric operation produces a numeric value. The permitted data types for operands of the numeric intrinsic operations are specified in 7.1.2.

The numeric operators and their interpretation in an expression are given in Table 7.3, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.3 Interpretation of the numeric intrinsic operators

Operator	Representing	Use of operator	Interpretation
**	Exponentiation	x_1**x_2	Raise x_1 to the power x_2
/	Division	x_1 / x_2	Divide x_1 by x_2
*	Multiplication	x_1 * x_2	Multiply x_1 by x_2
-	Subtraction	x_1 - x_2	Subtract x_2 from x_1
-	Negation	- x_2	Negate x_2
+	Addition	x_1 + x_2	Add x_1 and x_2
+	Identity	+ x_2	Same as x_2

The interpretation of a division depends on the data types of the operands (7.2.1.1).

If x_1 and x_2 are of type integer and x_2 has a negative value, the interpretation of x_1 ** x_2 is the same as the interpretation of 1/(x_1 ** ABS (x_2)), which is subject to the rules of integer division (7.2.1.1).

NOTE 7.31

For example, 2 ** (–3) has the value of 1/(2 ** 3), which is zero.

7.2.1.1 Integer division

One operand of type integer may be divided by another operand of type integer. Although the mathematical quotient of two integers is not necessarily an integer, Table 7.1 specifies that an expression involving the division operator with two operands of type integer is interpreted as an expression of type integer. The result of such an operation is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively.

NOTE 7.32

For example, the expression (–8) / 3 has the value (–2).

7.2.1.2 Complex exponentiation

In the case of a complex value raised to a complex power, the value of the operation x_1 ** x_2 is the principal value of $x_1^{x_2}$.

7.2.2 Character intrinsic operation

The character intrinsic operator // is used to concatenate two operands of type character with the same kind type parameter. Evaluation of the character intrinsic operation produces a result of type character.

The interpretation of the character intrinsic operator // when used to form an expression is given in Table 7.4, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.4 Interpretation of the character intrinsic operator //

Operator	Representing	Use of operator	Interpretation
//	Concatenation	x_1 // x_2	Concatenate x_1 with x_2

The result of the character intrinsic operation // is a character string whose value is the value of x_1 concatenated on the right with the value of x_2 and whose length is the sum of the lengths of x_1 and x_2. Parentheses used to specify the order of evaluation have no effect on the value of a character expression.

NOTE 7.33

For example, the value of ('AB' // 'CDE') // 'F' is the string 'ABCDEF'. Also, the value of 'AB' // ('CDE' // 'F') is the string 'ABCDEF'.

7.2.3 Relational intrinsic operations

A relational intrinsic operation is used to compare values of two operands using the relational intrinsic operators .LT., .LE., .GT., .GE., .EQ., .NE., <, <=, >, >=, ==, and /=. The permitted data types for operands of the relational intrinsic operators are specified in 7.1.2.

NOTE 7.34

As shown in Table 7.1, a relational intrinsic operator cannot be used to compare the value of an expression of a numeric type with one of type character or logical. Also, two operands of type logical cannot be compared, a complex operand may be compared with another numeric operand only when the operator is .EQ., .NE., ==, or /=, and two character operands cannot be compared unless they have the same kind type parameter value.

Evaluation of a relational intrinsic operation produces a result of type default logical.

The interpretation of the relational intrinsic operators is given in Table 7.5, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator. The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT., .LE., .GT., .GE., .EQ., and .NE., respectively.

Table 7.5 Interpretation of the relational intrinsic operators

Operator	Representing	Use of operator	Interpretation
.LT.	Less than	x_1 .LT. x_2	x_1 less than x_2
<	Less than	$x_1 < x_2$	x_1 less than x_2
.LE.	Less than or equal to	x_1 .LE. x_2	x_1 less than or equal to x_2
<=	Less than or equal to	$x_1 <= x_2$	x_1 less than or equal to x_2
.GT.	Greater than	x_1 .GT. x_2	x_1 greater than x_2
>	Greater than	$x_1 > x_2$	x_1 greater than x_2
.GE.	Greater than or equal to	x_1 .GE. x_2	x_1 greater than or equal to x_2
>=	Greater than or equal to	$x_1 >= x_2$	x_1 greater than or equal to x_2
.EQ.	Equal to	x_1 .EQ. x_2	x_1 equal to x_2
==	Equal to	$x_1 == x_2$	x_1 equal to x_2
.NE.	Not equal to	x_1 .NE. x_2	x_1 not equal to x_2
/=	Not equal to	x_1 /= x_2	x_1 not equal to x_2

A numeric relational intrinsic operation is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. A numeric relational intrinsic operation is interpreted as having the logical value false if the values of the operands do not satisfy the relation specified by the operator.

In the numeric relational operation

x_1 *rel-op* x_2

if the types or kind type parameters of x_1 and x_2 differ, their values are converted to the type and kind type parameter of the expression $x_1 + x_2$ before evaluation.

A character relational intrinsic operation is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator. A character relational intrinsic operation is interpreted as having the logical value false if the values of the operands do not satisfy the relation specified by the operator.

For a character relational intrinsic operation, the operands are compared one character at a time in order, beginning with the first character of each character operand. If the operands are of unequal length, the shorter operand is treated as if it were extended on the right with blanks to the length of the longer operand. If both x_1 and x_2 are of zero length, x_1 is equal to x_2; if every character of x_1 is the same as the character in the corresponding position in x_2, x_1 is equal to x_2. Otherwise, at the first position where the character operands differ, the character operand x_1 is considered to be less than x_2 if the character value of x_1 at this position precedes the value of x_2 in the collating sequence (4.3.2.1.1); x_1 is greater than x_2 if the character value of x_1 at this position follows the value of x_2 in the collating sequence.

NOTE 7.35

The collating sequence depends partially on the processor; however, the result of the use of the operators .EQ., .NE., ==, and /= does not depend on the collating sequence.

For nondefault character types, the blank padding character is processor dependent.

7.2.4 Logical intrinsic operations

A logical operation is used to express a logical computation. Evaluation of a logical operation produces a result of type logical. The permitted data types for operands of the logical intrinsic operations are specified in 7.1.2.

The logical operators and their interpretation when used to form an expression are given in Table 7.6, where x_1 denotes the operand to the left of the operator and x_2 denotes the operand to the right of the operator.

Table 7.6 Interpretation of the logical intrinsic operators

Operator	Representing	Use of operator	Interpretation
.NOT.	Logical negation	.NOT. x_2	True if x_2 is false
.AND.	Logical conjunction	x_1 .AND. x_2	True if x_1 and x_2 are both true
.OR.	Logical inclusive Disjunction	x_1 .OR. x_2	True if x_1 and/or x_2 is true
.NEQV.	Logical non-equivalence	x_1 .NEQV. x_2	True if either x_1 or x_2 is true, but not both
.EQV.	Logical equivalence	x_1 .EQV. x_2	True if both x_1 and x_2 are true or both are false

The values of the logical intrinsic operations are shown in Table 7.7.

Table 7.7 The values of operations involving logical intrinsic operators

x_1	x_2	.NOT. x_2	x_1 .AND. x_2	x_1 .OR. x_2	x_1 .EQV. x_2	x_1 .NEQV. x_2
true	true	false	true	true	true	false
true	false	true	false	true	false	true
false	true	false	false	true	false	true
false	false	true	false	false	true	false

7.3 Interpretation of defined operations

The interpretation of a defined operation is provided by the function that defines the operation. The type, type parameters, and interpretation of an expression that consists of a defined operation are independent of the type and type parameters of any larger expression in which it appears. The operators <, <=, >, >=, ==, and /= always have the same interpretations as the operators .LT., .LE., .GT., .GE., .EQ., and .NE., respectively.

7.3.1 Unary defined operation

A function defines the unary operation *op* x_2 if

1. (1) The function is specified with a FUNCTION (12.5.2.2) or ENTRY (12.5.2.5) statement that specifies one dummy argument d_2,
2. (2) An interface block (12.3.2.1) provides the function with a *generic-spec* of OPERATOR (*op*),
3. (3) The type of x_2 is the same as the type of dummy argument d_2,
4. (4) The type parameters, if any, of x_2 match those of d_2, and
5. (5) Either
   - (a) The function is elemental or
   - (b) The rank of x_2, and its shape if it is an array, match those of d_2.

7.3.2 Binary defined operation

A function defines the binary operation x_1 *op* x_2 if

(1) The function is specified with a FUNCTION (12.5.2.2) or ENTRY (12.5.2.5) statement that specifies two dummy arguments, d_1 and d_2,

(2) An interface block (12.3.2.1) provides the function with a *generic-spec* of OPERATOR (*op*),

(3) The types of x_1 and x_2 are the same as those of the dummy arguments d_1 and d_2, respectively,

(4) The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively, and

(5) Either

  (a) The function is elemental and x_1 and x_2 are conformable or

  (b) The ranks of x_1 and x_2, and their shapes if either or both are arrays, match those of d_1 and d_2, respectively.

7.4 Precedence of operators

There is a precedence among the intrinsic and extension operations implied by the general form in 7.1.1, which determines the order in which the operands are combined, unless the order is changed by the use of parentheses. This precedence order is summarized in Table 7.8.

Table 7.8 Categories of operations and relative precedence

Category of operation	Operators	Precedence
Extension	*defined-unary-op*	Highest
Numeric	**	.
Numeric	* or /	.
Numeric	unary + or –	.
Numeric	binary + or –	.
Character	//	.
Relational	.EQ., .NE., .LT., .LE., .GT., .GE., ==, /=, <, <=, >, >=	.
Logical	.NOT.	.
Logical	.AND.	.
Logical	.OR.	.
Logical	.EQV. or .NEQV.	.
Extension	*defined-binary-op*	Lowest

The precedence of a defined operation is that of its operator.

NOTE 7.36

For example, in the expression

```
-A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (–); therefore, the operands of the exponentiation operator are combined to form an expression that is used as the operand of the negation operator. The interpretation of the above expression is the same as the interpretation of the expression

```
- (A ** 2)
```

The general form of an expression (7.1.1) also establishes a precedence among operators in the same syntactic class. This precedence determines the order in which the operands are to be combined in determining the interpretation of the expression unless the order is changed by the use of parentheses.

NOTE 7.37

In interpreting a *level-2-expr* containing two or more binary operators + or –, each operand (*add-operand*) is combined from left to right. Similarly, the same left-to-right interpretation for a *mult-operand* in *add-operand*, as well as for other kinds of expressions, is a consequence of the general form. However, for interpreting a *mult-operand* expression when two or more exponentiation operators ** combine *level-1-expr* operands, each *level-1-expr* is combined from right to left.

For example, the expressions

```
2.1 + 3.4 + 4.9

2.1 * 3.4 * 4.9

2.1 / 3.4 / 4.9

2 ** 3 ** 4

'AB' // 'CD' // 'EF'
```

have the same interpretations as the expressions

```
(2.1 + 3.4) + 4.9

(2.1 * 3.4) * 4.9

(2.1 / 3.4) / 4.9

2 ** (3 ** 4)

('AB' // 'CD') // 'EF'
```

As a consequence of the general form (7.1.1), only the first *add-operand* of a *level-2-expr* may be preceded by the identity (+) or negation (–) operator. These formation rules do not permit expressions containing two consecutive numeric operators, such as A ** –B or A + –B. However, expressions such as A ** (–B) and A + (–B) are permitted. The rules do allow a binary operator or an intrinsic unary operator to be followed by a defined unary operator, such as:

```
A * .INVERSE. B

- .INVERSE. (B)
```

As another example, in the expression

```
A .OR. B .AND. C
```

the general form implies a higher precedence for the .AND. operator than for the .OR. operator; therefore, the interpretation of the above expression is the same as the interpretation of the expression

```
A .OR. (B .AND. C)
```

NOTE 7.38

An expression may contain more than one category of operator. The logical expression

```
L .OR. A + B >= C
```

where A, B, and C are of type real, and L is of type logical, contains a numeric operator, a relational operator, and a logical operator. This expression would be interpreted the same as the expression

```
L .OR. ((A + B) >= C)
```

NOTE 7.38 ***(Continued)***

For example, if

(1) The operator ** is extended to type logical,
(2) The operator .STARSTAR. is defined to duplicate the function of ** on type real,
(3) .MINUS. is defined to duplicate the unary operator –, and
(4) L1 and L2 are type logical and X and Y are type real,

then in precedence: L1 ** L2 is higher than X * Y; X * Y is higher than X .STARSTAR. Y; and .MINUS. X is higher than –X.

7.5 Assignment

Execution of an assignment statement causes a variable to become defined or redefined. Execution of a pointer assignment statement causes a pointer to become associated with a target or causes its pointer association status to become disassociated or undefined. Execution of a WHERE statement or WHERE construct masks the evaluation of expressions and assignment of values in array assignment statements according to the value of a logical array expression. Execution of a FORALL statement or FORALL construct controls the assignment to elements of arrays by using a set of index variables and a mask expression.

7.5.1 Assignment statement

A variable may be defined or redefined by execution of an assignment statement.

7.5.1.1 General form

R735 *assignment-stmt* **is** *variable = expr*

where *variable* is defined in R601 and *expr* is defined in R723.

Constraint: A variable in an *assignment-stmt* shall not be a whole assumed-size array.

NOTE 7.39

Examples of an assignment statement are:

```
A = 3.5 + X * Y
I = INT (A)
```

An assignment statement is either intrinsic or defined.

7.5.1.2 Intrinsic assignment statement

An **intrinsic assignment statement** is an assignment statement where the shapes of *variable* and *expr* conform and where

(1) The types of *variable* and *expr* are intrinsic, as specified in Table 7.9 for assignment, or
(2) The types of *variable* and *expr* are of the same derived type and no defined assignment exists for objects of this type.

A **numeric intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of numeric type. A **character intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type character and have the same kind type parameter. A **logical intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of type logical. A **derived-type intrinsic assignment statement** is an intrinsic assignment statement for which *variable* and *expr* are of the same derived type, and there is no accessible interface block with an ASSIGNMENT (=) specifier for objects of this derived type.

An **array intrinsic assignment statement** is an intrinsic assignment statement for which *variable* is an array. The *variable* shall not be a many-one array section (6.2.2.3.2).

Table 7.9 Type conformance for the intrinsic assignment statement

Type of *variable*	Type of *expr*
integer	integer, real, complex
real	integer, real, complex
complex	integer, real, complex
character	character of the same kind type parameter as *variable*
logical	logical
derived type	same derived type as *variable*

7.5.1.3 Defined assignment statement

A **defined assignment statement** is an assignment statement that is not an intrinsic assignment statement, and is defined by a subroutine and an interface block (12.3.2.1) that specifies ASSIGNMENT (=). A **defined elemental assignment statement** is a defined assignment statement for which the subroutine is elemental (12.7).

7.5.1.4 Intrinsic assignment conformance rules

For an intrinsic assignment statement, *variable* and *expr* shall conform in shape, and if *expr* is an array, *variable* shall also be an array. The types of *variable* and *expr* shall conform with the rules of Table 7.9.

If *variable* is a pointer, it shall be associated with a definable target such that the type, type parameters, and shape of the target and *expr* conform.

For a numeric intrinsic assignment statement, *variable* and *expr* may have different numeric types or different kind type parameters, in which case the value of *expr* is converted to the type and kind *variable* according to the rules of Table 7.10.

Table 7.10 Numeric conversion and the assignment statement

Type of *variable*	Value Assigned
integer	INT (*expr*, KIND = KIND (*variable*))
real	REAL (*expr*, KIND = KIND (*variable*))
complex	CMPLX (*expr*, KIND = KIND (*variable*))

Note: The functions INT, REAL, CMPLX, and KIND are the generic functions defined in 13.14.

For a logical intrinsic assignment statement, *variable* and *expr* may have different kind type parameters, in which case the value of *expr* is converted to the kind type parameter of *variable*.

For a character intrinsic assignment statement, *variable* and *expr* shall have the same kind type parameter value, but may have different character length parameters in which case the conversion of *expr* to the length of *variable* is as follows:

(1) If the length of *variable* is less than that of *expr*, the value of *expr* is truncated from the right until it is the same length as *variable*.

(2) If the length of *variable* is greater than that of *expr*, the value of *expr* is extended on the right with blanks until it is the same length as *variable*.

NOTE 7.40

For nondefault character types, the blank padding character is processor dependent.

7.5.1.5 Interpretation of intrinsic assignments

Execution of an intrinsic assignment causes, in effect, the evaluation of the expression *expr* and all expressions within *variable* (7.1.7), the possible conversion of *expr* to the type and type parameters of *variable* (Table 7.10), and the definition of *variable* with the resulting value. The execution of the assignment shall have the same effect as if the evaluation of all operations in *expr* and *variable* occurred before any portion of *variable* is defined by the assignment. The evaluation of expressions within *variable* shall neither affect nor be affected by the evaluation of *expr*. No value is assigned to *variable* if *variable* is of type character and zero length, or is an array of size zero.

If *variable* is a pointer, the value of *expr* is assigned to the target of *variable*.

Both *variable* and *expr* may contain references to any portion of *variable*.

NOTE 7.41

For example, in the character intrinsic assignment statement:

```
STRING (2:5) = STRING (1:4)
```

the assignment of the first character of STRING to the second character does not affect the evaluation of STRING (1:4).

For example, if the value of STRING prior to the assignment was 'ABCDEF', the value following the assignment is 'AABCDF'.

If *expr* in an intrinsic assignment is a scalar and *variable* is an array, the *expr* is treated as if it were an array of the same shape as *variable* with every element of the array equal to the scalar value of *expr*.

When *variable* in an intrinsic assignment is an array, the assignment is performed element-by-element on corresponding array elements of *variable* and *expr*.

NOTE 7.42

For example, if A and B are arrays of the same shape, the array intrinsic assignment

```
A = B
```

assigns the corresponding elements of B to those of A; that is, the first element of B is assigned to the first element of A, the second element of B is assigned to the second element of A, etc.

The processor may perform the element-by-element assignment in any order.

NOTE 7.43

For example, the following program segment results in the values of the elements of array X being reversed:

```
REAL X (10)
   ...
X (1:10) = X (10:1:-1)
```

A derived-type intrinsic assignment is performed as if each component of *expr* were assigned to the corresponding component of *variable* using pointer assignment (7.5.2) for pointer components, and intrinsic assignment for nonpointer components. The processor may perform the component-by-component assignment in any order or by any means that has the same effect.

NOTE 7.44

For an example of a derived-type intrinsic assignment statement, if C and D are of the same derived type with a pointer component P and nonpointer components S, T, U, and V of type integer, logical, character, and another derived type, respectively, the intrinsic

```
C = D
```

NOTE 7.44 *(Continued)*

pointer assigns D % P to C % P and assigns D % S to C % S using the numeric intrinsic assignment statement, D % T to C % T using the logical intrinsic assignment statement, D % U to C % U using the character intrinsic assignment statement, and D % V to C % V using the derived-type intrinsic assignment statement.

When *variable* is a subobject, the assignment does not affect the definition status or value of other parts of the object. For example, if *variable* is an array section, the assignment does not affect the definition status or value of the elements of the parent array not specified by the array section.

7.5.1.6 Interpretation of defined assignment statements

The interpretation of a defined assignment is provided by the subroutine that defines the operation.

A subroutine defines the defined assignment $x_1 = x_2$ if

(1) The subroutine is specified with a SUBROUTINE (12.5.2.3) or ENTRY (12.5.2.5) statement that specifies two dummy arguments, d_1 and d_2,

(2) An interface block (12.3.2.1) provides the subroutine with a *generic-spec* of ASSIGNMENT (=),

(3) The types of x_1 and x_2 are the same as those of the dummy arguments d_1 and d_2, respectively,

(4) The type parameters, if any, of x_1 and x_2 match those of d_1 and d_2, respectively, and

(5) Either

  (a) The subroutine is elemental and either x_1 and x_2 have the same shape or x_2 is scalar or

  (b) The ranks of x_1 and x_2, and their shapes if either or both are arrays, match those of d_1 and d_2, respectively.

The types of x_1 and x_2 shall not both be numeric, both be logical, or both be character with the same kind type parameter value.

If the defined assignment is an elemental assignment and the *variable* in the assignment is an array, the defined assignment is performed element-by-element, in any order, on corresponding elements of *variable* and *expr*. If *expr* is a scalar, it is treated as if it were an array of the same shape as *variable* with every element of the array equal to the scalar value of *expr*.

NOTE 7.45

The rules of defined assignment (12.3.2.1.2), procedure references (12.4), subroutine references (12.4.3), and elemental subroutine arguments (12.7.3) ensure that the defined assignment has the same effect as if the evaluation of all operations in x_2 and x_1 occurs before any portion of x_1 is defined.

7.5.2 Pointer assignment

Pointer assignment causes a pointer to become associated with a target or causes its pointer association status to become disassociated or undefined.

R736	*pointer-assignment-stmt*	**is**	*pointer-object* => *target*
R737	*target*	**is**	*variable*
		or	*expr*

Constraint: The *pointer-object* shall have the POINTER attribute.

Constraint: The *variable* shall have the TARGET attribute or be a subobject of an object with the TARGET attribute, or it shall have the POINTER attribute.

Constraint: The *target* shall be of the same type, kind type parameters, and rank as the pointer.

Constraint: The *target* shall not be an array section with a vector subscript.

Constraint: The *expr* shall deliver a pointer result.

The *target* shall have the same type parameters as the *pointer-object*. If the *target* is not a pointer, the pointer assignment statement associates the *pointer-object* with the *target*. If the *target* is a pointer that is associated, the *pointer-object* is associated with the same object as the *target*. If the *target* is a pointer that is disassociated or a reference to the NULL intrinsic function, the *pointer-object* becomes disassociated. If the *target* is a pointer with undefined association status, the *pointer-object* acquires an undefined association status.

Any previous association between the *pointer-object* and a target is broken.

Pointer assignment for a pointer component of a structure also may take place by execution of a derived-type intrinsic assignment statement (7.5.1.5) or a defined assignment statement (7.5.1.6).

In addition to pointer assignment, a pointer becomes associated with a target by allocation of the pointer.

A pointer shall not be referenced or defined unless it is associated with a target that may be referenced or defined.

NOTE 7.46

The following are examples of pointer assignment statements.

```
NEW_NODE % LEFT => CURRENT_NODE

SIMPLE_NAME => TARGET_STRUCTURE % SUBSTRUCT % COMPONENT

PTR => NULL ( )

ROW => MAT2D (N, :)

WINDOW => MAT2D (I-1:I+1, J-1:J+1)

POINTER_OBJECT => POINTER_FUNCTION (ARG_1, ARG_2)

EVERY_OTHER => VECTOR (1:N:2)
```

7.5.3 Masked array assignment - WHERE

The masked array assignment is used to mask the evaluation of expressions and assignment of values in array assignment statements, according to the value of a logical array expression.

7.5.3.1 General form of the masked array assignment

A **masked array assignment** is either a WHERE statement or WHERE construct.

R738 *where-stmt* **is** WHERE (*mask-expr*) *where-assignment-stmt*

R739 *where-construct* **is** *where-construct-stmt*
    [*where-body-construct*] ...
  [*masked-elsewhere-stmt*
    [*where-body-construct*] ...] ...
  [*elsewhere-stmt*
    [*where-body-construct*] ...]
  end-where-stmt

R740 *where-construct-stmt* **is** [*where-construct-name*:] WHERE (*mask-expr*)

R741 *where-body-construct* **is** *where-assignment-stmt*
or *where-stmt*
or *where-construct*

R742 *where-assignment-stmt* **is** *assignment-stmt*

R743 *mask-expr* **is** *logical-expr*

R744 *masked-elsewhere-stmt* **is** ELSEWHERE (*mask-expr*) [*where-construct-name*]

R745 *elsewhere-stmt* **is** ELSEWHERE [*where-construct-name*]

R746 *end-where-stmt* **is** END WHERE [*where-construct-name*]

Constraint: A *where-assignment-stmt* that is a defined assignment shall be elemental.

Constraint: If the *where-construct-stmt* is identified by a *where-construct-name*, the corresponding *end-where-stmt* shall specify the same *where-construct-name*. If the *where-construct-stmt* is not identified by a *where-construct-name*, the corresponding *end-where-stmt* shall not specify a *where-construct-name*. If an *elsewhere-stmt* or a *masked-elsewhere-stmt* is identified by a *where-construct-name*, the corresponding *where-construct-stmt* shall specify the same *where-construct-name*.

If a *where-construct* contains a *where-stmt*, a *masked-elsewhere-stmt*, or another *where-construct* then each *mask-expr* within the *where-construct* shall have the same shape. In each *where-assignment-stmt*, the *mask-expr* and the *variable* being defined shall be arrays of the same shape.

NOTE 7.47

Examples of a masked array assignment are:

```
WHERE (TEMP > 100.0) TEMP = TEMP - REDUCE_TEMP

WHERE (PRESSURE <= 1.0)
   PRESSURE = PRESSURE + INC_PRESSURE
   TEMP = TEMP - 5.0
ELSEWHERE
   RAINING = .TRUE.
END WHERE
```

7.5.3.2 Interpretation of masked array assignments

When a WHERE statement or a *where-construct-stmt* is executed, a control mask is established. In addition, when a WHERE construct statement is executed, a pending control mask is established. If the statement does not appear as part of a *where-body-construct*, the *mask-expr* of the statement is evaluated, and the control mask is established to be the value of *mask-expr*. The pending control mask is established to have the value **.**NOT**.** *mask-expr* upon execution of a WHERE construct statement that does not appear as part of a *where-body-construct*. The *mask-expr* is evaluated only once.

Each statement in a WHERE construct is executed in sequence.

Upon execution of a *masked-elsewhere-stmt*, the following actions take place in sequence:

(1) The control mask m_c is established to have the value of the pending control mask.

(2) The pending control mask is established to have the value m_c **.**AND**.** (**.**NOT**.** *mask-expr).*

(3) The control mask m_c is established to have the value m_c **.**AND**.** *mask-expr*.

The *mask-expr* is evaluated only once.

Upon execution of an ELSEWHERE statement, the control mask is established to have the value of the pending control mask. No new pending control mask value is established.

Upon execution of an ENDWHERE statement, the control mask and pending control mask are established to have the values they had prior to the execution of the corresponding WHERE construct statement. Following the execution of a WHERE statement that appears as a *where-body-construct*, the control mask is established to have the value it had prior to the execution of the WHERE statement.

NOTE 7.48

The establishment of control masks and the pending control mask is illustrated with the following example:

```
WHERE(cond1)          ! Statement 1
. . .
ELSEWHERE(cond2)      ! Statement 2
  . . .
ELSEWHERE             ! Statement 3
. . .
END WHERE
```

Following execution of statement 1, the control mask has the value cond1 and the pending control mask has the value .NOT. cond1. Following execution of statement 2, the control mask has the value (.NOT. cond1) .AND. cond2 and the pending control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). Following execution of statement 3, the control mask has the value (.NOT. cond1) .AND. (.NOT. cond2). The false condition values are propagated through the execution of the masked ELSEWHERE statement.

Upon execution of a WHERE statement or a WHERE construct statement that is part of a *where-body-construct*, the pending control mask is established to have the value m_c .AND. (.NOT. *mask-expr*). The control mask is then established to have the value m_c .AND. *mask-expr*. The *mask-expr* is evaluated only once.

If a nonelemental function reference occurs in the *expr* or *variable* of a *where-assignment-stmt* or in a *mask-expr*, the function is evaluated without any masked control; that is, all of its argument expressions are fully evaluated and the function is fully evaluated. If the result is an array and the reference is not within the argument list of a nonelemental function, elements corresponding to true values in the control mask are selected for use in evaluating the *expr*, *variable* or *mask-expr*.

If an elemental operation or function reference occurs in the *expr* or *variable* of a *where-assignment-stmt* or in a *mask-expr*, and is not within the argument list of a nonelemental function reference, the operation is performed or the function is evaluated only for the elements corresponding to true values of the control mask.

If an array constructor appears in a *where-assignment-stmt* or in a *mask-expr*, the array constructor is evaluated without any masked control and then the *where-assignment-stmt* is executed or the *mask-expr* is evaluated.

When a *where-assignment-stmt* is executed, the values of *expr* that correspond to true values of the control mask are assigned to the corresponding elements of *variable*.

A statement that is part of a *where-body-construct* shall not be a branch target statement. The value of the control mask is established by the execution of a WHERE statement, a WHERE construct statement, an ELSEWHERE statement, a masked ELSEWHERE statement, or an ENDWHERE statement. Subsequent changes to the value of entities in a *mask-expr* have no effect on the value of the control mask. The execution of a function reference in the mask expression of a WHERE statement is permitted to affect entities in the assignment statement. Execution of an END WHERE has no effect.

NOTE 7.49

Examples of function references in masked array assignments are:

```
WHERE (A > 0.0)
   A = LOG (A)           ! LOG is invoked only for positive elements.
   A = A / SUM (LOG (A))  ! LOG is invoked for all elements
                          ! because SUM is transformational.
END WHERE
```

7.5.4 FORALL

FORALL constructs and statements are used to control the execution of assignment and pointer assignment statements with selection by sets of index values and an optional mask expression.

7.5.4.1 The FORALL Construct

The FORALL construct allows multiple assignments, masked array (WHERE) assignments, and nested FORALL constructs and statements to be controlled by a single *forall-triplet-spec-list* and *scalar-mask*.

R747 *forall-construct* **is** *forall-construct- stmt*
    [*forall-body-construct*] ...
    end-forall-stmt

R748 *forall-construct-stmt* **is** [*forall-construct-name* :] FORALL *forall-header*

R749 *forall-header* **is** (*forall-triplet-spec-list* [, *scalar-mask-expr*])

R750 *forall-triplet-spec* **is** *index-name* = *subscript* : *subscript* [: *stride*]

R617 *subscript* **is** *scalar-int-expr*

R603 *stride* **is** *scalar-int-expr*

R751 *forall-body-construct* **is** *forall-assignment-stmt*
or *where-stmt*
or *where-construct*
or *forall-construct*
or *forall-stmt*

R752 *forall-assignment-stmt* **is** *assignment-stmt*
or *pointer-assignment-stmt*

R753 *end-forall-stmt* **is** END FORALL [*forall-construct-name*]

Constraint: If the *forall-construct-stmt* has a *forall-construct-name*, the *end-forall-stmt* shall have the same *forall-construct-name*. If the *end-forall-stmt* has a *forall-construct-name*, the *forall-construct-stmt* shall have the same *forall-construct-name*.

Constraint: The *scalar-mask-expr* shall be scalar and of type logical.

Constraint: Any procedure referenced in the *scalar-mask-expr*, including one referenced by a defined operation, shall be a pure procedure (12.6).

Constraint: The *index-name* shall be a named scalar variable of type integer.

Constraint: A *subscript* or *stride* in a *forall-triplet-spec* shall not contain a reference to any *index-name* in the *forall-triplet-spec-list* in which it appears.

Constraint: A statement in a *forall-body-construct* shall not define an *index-name* of the *forall-construct*.

Constraint: Any procedure referenced in a *forall-body-construct*, including one referenced by a defined operation or assignment, shall be a pure procedure.

Constraint: A *forall-body-construct* shall not be a branch target.

NOTE 7.50

An example of a FORALL construct is:

```
REAL :: A(10, 10), B(10, 10) = 1.0
. . .
FORALL (I = 1:10, J = 1:10, B(I, J) /= 0.0)
   A(I, J) = REAL (I + J - 2)
   B(I, J) = A(I, J) + B(I, J) * REAL (I * J)
END FORALL
```

NOTE 7.51

An assignment statement that is a FORALL body construct may be a scalar or array assignment statement, or a defined assignment statement. The variable being defined will normally use each index name in the *forall-triplet-spec-list*. For example

```
FORALL (I = 1:N, J = 1:N)
   A(:, I, :, J) = 1.0 / REAL(I + J - 1)
END FORALL
```

broadcasts scalar values to rank-two subarrays of A.

NOTE 7.52

An example of a FORALL construct containing a pointer assignment statement is:

```
TYPE ELEMENT
   REAL ELEMENT_WT
   CHARACTER (32), POINTER :: NAME
END TYPE ELEMENT
TYPE(ELEMENT) CHART(200)
REAL WEIGHTS (1000)
CHARACTER (32), TARGET :: NAMES (1000)
. . .
FORALL (I = 1:200, WEIGHTS (I + N - 1) > .5)
   CHART(I) % ELEMENT_WT = WEIGHTS (I + N - 1)
   CHART(I) % NAME => NAMES (I + N - 1)
END FORALL
```

The results of this FORALL construct cannot be achieved with a WHERE construct because a pointer assignment statement is not permitted in a WHERE construct.

An *index-name* in a *forall-construct* has a scope of the construct (14.1.3). It is a scalar variable that has the type and type parameters that it would have if it were the name of a variable in the scoping unit that includes the FORALL, and this type shall be integer type; it has no other attributes.

NOTE 7.53

The use of *index-name* variables in a FORALL construct does not affect variables of the same name, for example:

```
INTEGER :: X = -1
REAL A(5, 4)
J = 100
. . .
FORALL (X = 1:5, J = 1:4)
   A (X, J) = J
END FORALL
```

After execution of the FORALL, the variables X and J have the values -1 and 100 and A has the value

```
      1 2 3 4
      1 2 3 4
      1 2 3 4
      1 2 3 4
      1 2 3 4
```

7.5.4.2 Execution of the FORALL construct

There are three stages in the execution of a FORALL construct:

(1) Determination of the values for *index-name* variables,

(2) Evaluation of the *scalar-mask-expr*, and

(3) Execution of the FORALL body constructs.

7.5.4.2.1 Determination of the values for *index-name* variables

The subscript and stride expressions in the *forall-triplet-spec-list* are evaluated. These expressions may be evaluated in any order. The set of values that a particular *index-name* variable assumes is determined as follows:

(1) The lower bound m_1, the upper bound m_2, and the stride m_3 are of type integer with the same kind type parameter as the *index-name*. Their values are established by evaluating the first subscript, the second subscript, and the stride expressions, respectively, including, if necessary, conversion to the kind type parameter of the *index-name* according to the rules for numeric conversion (Table 7.10). If a stride does not appear, m_3 has the value 1. The value m_3 shall not be zero.

(2) Let the value of max be $(m_2 - m_1 + m_3)/m_3$. If $max \leq 0$ for some *index-name*, the execution of the construct is complete. Otherwise, the set of values for the *index-name* is

$$m_1 + (k-1) \times m_3 \qquad \text{where } k = 1, 2, ..., max\,.$$

The set of combinations of *index-name* values is the Cartesian product of the sets defined by each triplet specification. An *index-name* becomes defined when this set is evaluated.

NOTE 7.54

The *stride* may be positive or negative; the FORALL body constructs are executed as long as max > 0. For the *forall-triplet-spec*

```
I = 10:1:-1
```

max has the value 10

7.5.4.2.2 Evaluation of the *scalar-mask-expr*

The *scalar-mask-expr*, if any, is evaluated for each combination of *index-name* values. If the *scalar-mask-expr* is not present, it is as if it were present with the value .TRUE.. The *index-name* variables may be primaries in the *scalar-mask-expr*.

The **active combination of *index-name* value**s is defined to be the subset of all possible combinations (7.5.4.2.1) for which the *scalar-mask-expr* has the value .TRUE..

NOTE 7.55

The *index-name* variables may appear in the mask, for example

```
FORALL (I=1:10, J=1:10, A(I) > 0.0 .AND. B(J) < 1.0)
. . .
```

7.5.4.2.3 Execution of the FORALL body constructs

The *forall-body-construct*s are executed in the order in which they appear. Each construct is executed for all active combinations of the *index-name* values with the following interpretation:

Execution of a *forall-assignment-stmt* that is an *assignment-stmt* causes the evaluation of *expr* and all expressions within *variable* for all active combinations of *index-name* values. These evaluations may be done in any order. After all these evaluations have been performed, each *expr* value is assigned to the corresponding *variable*. The assignments may occur in any order.

Execution of a *forall-assignment-stmt* that is a *pointer-assignment-stmt* causes the evaluation of all expressions within *target* and *pointer-object*, the determination of any pointers within *pointer-object*, and the determination of the target for all active combinations of *index-name* values. These evaluations may be done in any order. After all these evaluations have been performed, each *pointer-object* is associated with the corresponding *target*. These associations may occur in any order.

In a *forall-assignment-stmt*, a defined assignment subroutine shall not reference any *variable* that becomes defined or *pointer-object* that becomes associated by the statement.

NOTE 7.56

The following FORALL construct contains two assignment statements. The assignment to array B uses the values of array A computed in the previous statement, not the values A had prior to execution of the FORALL.

```
   FORALL (I = 2:N-1, J = 2:N-1 )
      A (I, J) = A(I, J-1) + A(I,J+1) + A(I-1,J) + A(I+1, J)
      B (I, J) = 1.0 / A(I, J)
   END FORALL
```

Computations that would otherwise cause error conditions can be avoided by using an appropriate *scalar-mask-expr* that limits the active combinations of the *index-name* values. For example:

```
   FORALL (I = 1:N, Y(I) .NE. 0.0)
      X(I) = 1.0 / Y(I)
   END FORALL
```

Each statement in a *where-construct* (7.5.3) within a *forall-construct* is executed in sequence. When a *where-stmt*, *where-construct-stmt* or *masked-elsewhere-stmt* is executed, the statement's *mask-expr* is evaluated for all active combinations of *index-name* values as determined by the outer *forall-construct*s, masked by any control mask corresponding to outer *where-construct*s. Any *where-assignment-stmt* is executed for all active combinations of *index-name* values, masked by the control mask in effect for the *where-assignment-stmt*.

NOTE 7.57

This FORALL construct contains a WHERE statement and an assignment statement.

```
   INTEGER A(5,4), B(5,4)
   FORALL ( I = 1:5 )
      WHERE ( A(I,:) .EQ. 0 ) A(I,:) = I
      B (I,:) = I / A(I,:)
   END FORALL
```

When executed with the input array

```
          0  0  0  0
          1  1  1  0
   A  =   2  2  0  2
          1  0  2  3
          0  0  0  0
```

the results will be

```
          1  1  1  1               1  1  1  1
          1  1  1  2               2  2  2  1
   A  =   2  2  3  2        B  =   1  1  1  1
          1  4  2  3               4  1  2  1
          5  5  5  5               1  1  1  1
```

For an example of a FORALL construct containing a WHERE construct with an ELSEWHERE statement, see C.4.5.

Execution of a *forall-stmt* or *forall-construct* causes the evaluation of the *subscript* and *stride* expressions in the *forall-triplet-spec-list* for all active combinations of the *index-name* values of the outer FORALL construct. The set of combinations of *index-name* values for the inner FORALL is the union of the sets defined by these bounds and strides for each active combination of the outer *index-name* values; it also includes the outer *index-name* values. The *scalar-mask-expr* is then evaluated for all combinations of the *index-name* values of the inner construct to produce a set of active combinations for the inner construct. If there is no *scalar-mask-expr*, it is as if it were present with the value .TRUE.. Each statement in the inner FORALL is then executed for each active combination of the *index-name* values.

NOTE 7.58

This FORALL construct contains a nested FORALL construct. It assigns the transpose of the lower triangle of array A (the section below the main diagonal) to the upper triangle of A.

```
   INTEGER A (3, 3)
   FORALL (I = 1:N-1 )
      FORALL ( J=I+1:N )
         A(I,J) = A(J,I)
      END FORALL
   END FORALL
```

If prior to execution N = 3 and

```
         0  3  6
   A  =  1  4  7
         2  5  8
```

then after execution

```
         0  1  2
   A  =  1  4  5
         2  5  8
```

7.5.4.3 The FORALL statement

The FORALL statement allows a single assignment statement or pointer assignment to be controlled by a set of index values and an optional mask expression.

R754 *forall-stmt* **is** FORALL *forall-header forall-assignment-stmt*

A FORALL statement is equivalent to a FORALL construct containing a single *forall-body-construct* that is a *forall-assignment-stmt*.

The scope of an *index-name* in a *forall-stmt* is the statement itself (14.1.3).

NOTE 7.59

Examples of FORALL statements are:

```
   FORALL (I=1:N) A(I,I) = X(I)
```

This statement assigns the elements of vector X to the elements of the main diagonal of matrix A.

```
   FORALL (I = 1:N, J = 1:N)  X(I,J) = 1.0 / REAL (I+J-1)
```

Array element X(I,J) is assigned the value (1.0 / REAL (I+J-1)) for values of I and J between 1 and N, inclusive.

```
   FORALL (I=1:N, J=1:N, Y(I,J) /= 0 .AND. I /= J)  X(I,J) = 1.0 / Y(I,J)
```

This statement takes the reciprocal of each nonzero off-diagonal element of array Y(1:N, 1:N) and assigns it to the corresponding element of array X. Elements of Y that are zero or on the diagonal do not participate, and no assignments are made to the corresponding elements of X.

The results from the execution of the example in Note 7.58 could be obtained with a single FORALL statement:

```
  FORALL ( I = 1:N-1, J=1:N, J > I )  A(I,J) = A(J,I)
```

For more examples of FORALL statements, see C.4.6.

7.5.4.4 Restrictions on FORALL constructs and statements

A many-to-one assignment is more than one assignment to the same object or subobject, or association of more than one target with the same pointer, whether the object is referenced directly or indirectly through a pointer. A many-to-one assignment shall not occur within a single statement in a FORALL construct or statement. It is possible to assign or pointer assign to the same object in different assignment statements in a FORALL construct.

NOTE 7.60

The appearance of each *index-name* in the identification of the left-hand side of an assignment statement is helpful in eliminating many-to-one assignments, but it is not sufficient to guarantee there will be none. For example, the following is allowed

```
FORALL (I = 1:10)
   A (INDEX (I)) = B(I)
END FORALL
```

if and only if INDEX(1:10) contains no repeated values.

Within the scope of a FORALL construct, a nested FORALL statement or FORALL construct shall not have the same *index-name*. The *forall-header* expressions within a nested FORALL may depend on the values of outer *index-name* variables.