Chapter 5

Data Declarations and Specifications

Every object has a type and a rank as well as other properties that determine the ways in which objects of that type may be used. Collectively, these properties (including the type) are called the *attributes* of the object. The type of a named object either is determined implicitly by the first letter of its name or is specified explicitly in a type declaration statement. The implicit typing rules in effect within a program unit may be changed using the `IMPLICIT` statement. An attribute may be specified for an object by a type declaration statement or by using separate attribute specification statements.

5.1 Scalars and Arrays

An object or an expression is either a scalar or an array, and has a numeric attribute called its *rank*. A *scalar* is an object that represents a single value; if defined, its value is a single value from the set of values that characterize its data type. The rank of a scalar is zero.

An *array* is an ordered set of scalar data, all of the same type, arranged in an *n*-dimensional rectangular pattern. A named array has one subscript for each dimension of the pattern in which its elements are arranged. The scalar data that make up an array are the *array elements*. The pattern may have up to 31 dimensions, and any *extent* (size) in each dimension. The rank of the array is the number of its dimensions, and its *size* is the total number of elements, which is the product of the extents. An array may have a size of zero. An *array section* is a subset of the elements of a named array and is itself an array. Array objects may be of any type. An *array operation* is a reference or an update to a whole array or a subarray of an array rather than an individual element. (An array section is an array operation.)

The *shape* of an array is determined by its rank and its extent in each dimension, and may itself be represented as a rank-one array whose elements are the extents. For example, the array

```
DIMENSION X(200,300,400)
```


Copyright © 1989–1994 Thinking Machines Corporation

has a shape vector of [200,300,400]. All named arrays must be declared, and the rank of a named array is specified in its declaration. The rank of a named array is constant and the extents may be constant also. (However, the extents of allocatable arrays, dummy argument arrays, and automatic arrays may differ from one invocation of a procedure to another. See Chapter 6 for information about allocatable arrays. See Chapter 12 for information about procedures and dummy arguments.) Note that arrays of the same shape may have different lower and upper bounds in each dimension. For example,

```
DIMENSION A(1:10,3:5), B(11:20,-1:1)
```

declares the arrays A and B, both with shape [10,3].

Two arrays are *conformable* if they have the same shape. A scalar is conformable with any array. The elemental intrinsic functions (Chapter 17) as well as the numeric, relational, and logical operators may be applied to conformable array objects in an expression. Such array operations are performed element by element to produce a resultant array conformable with the array operands. Element-by-element operation means that corresponding elements of the operand arrays are involved in a "scalar-like" operation to produce the corresponding element in the result array, and all such element operations may be performed concurrently, or nearly concurrently, in arbitrary order.

A *whole array* is an array name with no subscripts. The name may refer to a constant array expression or an array variable. The array name represents all elements of the array. Except in some storage association contexts (see Chapter 8), no particular order of reference to the elements of an array is implied by the appearance of a whole array name.

5.2 Array Specifiers

The rank and shape of an array are given by an *array specifier* (R5-1) appearing in a type declaration statement (R5-10), or a DIMENSION statement (R5-52), or a COMMON statement (R8-1) declaring the array.

An *array-spec* is one of: R5-1

- *explicit-shape-array-spec*
- *assumed-shape-array-spec*
- *deferred-shape-array-spec*
- *assumed-size-array-spec*

Explicit-shape arrays can be declared globally (in a common block) or locally, while assumed-shape and assumed-size arrays can only be declared as dummy arguments to a

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

procedure. Deferred-shape arrays can be declared globally, locally, or as dummy arguments. (Procedures and dummy arguments are discussed in Chapter 12.)

- Explicit-shape array declarations specify the bounds in every dimension of an array: for global (common) arrays, the bounds must be constant; for local arrays, the bounds may be fixed at procedure entry.
- Assumed-shape array declarations specify the rank of a dummy array (see Chapter 12), but the size in each dimension is determined by the actual array argument passed in.
- Deferred-shape array declarations specify the rank of an allocatable array (see Chapter 6). The array's extents are determined when space is allocated for the array.
- Assumed-size array declarations specify the rank of a dummy array (see Chapter 12) and the size of each dimension but the last: the actual bound in the last dimension must be determined from program context.

The four kinds of array declarations are distinguished by the syntax of their array specifiers. The syntax and meaning of the four kinds of array specifiers are given in the following sections. Examples of type declaration statements and attribute specification statements using the different kinds of array specifiers are given in Sections 5.3, 5.4, and 5.5.

5.2.1 Explicit-Shape Arrays

An *explicit-shape array* is a named array that is declared with an *explicit-shape-array-spec* (R5-2), in which each *explicit-shape-spec* (R5-3) specifies the upper bound (and possibly the lower bound) in every dimension of the array. Each *explicit-shape-spec* specifies explicit values for the extent of the array in each dimension. An explicit-shape array may be a CM array or a front-end array.

An *explicit-shape-array-spec* is: R5-2
explicit-shape-spec [, explicit-shape-spec]...

An *explicit-shape-spec* is: R5-3
[lower-bound :] upper-bound

A *lower-bound* is: R5-4
specification-expr

An *upper-bound* is: R5-5
specification-expr

Copyright © 1989–1994 Thinking Machines Corporation

- The maximum number of *explicit-shape-spec*s in an *explicit-shape-array-spec* is 31.
- An explicit-shape array whose bounds are nonconstant expressions must be a dummy argument, a function result, or a local (automatic) array of a procedure.

The values of each *lower-bound* and *upper-bound* expression determine the bounds of the array along the given dimension and hence the extent of the array in that dimension. The value of a lower bound or an upper bound may be positive, negative, or zero. The subscript range of the array in a particular dimension is the set of integer values between and including the lower and upper bounds, provided the upper bound is not less than the lower bound. If the upper bound is less than the lower bound, the range is empty, the extent in that dimension is zero, and the array is of zero size. If a *lower-bound* is omitted, its default value is 1.

The rank of an explicit-shape array is the number of *upper-bound*s specified in its declaration.

Any of the bound expressions of an *explicit-shape-spec* may be nonconstant expressions provided that the expressions depend only on entities defined previously in the specification part of the subprogram in which the array is declared (see Specification Expressions, Section 7.12). Arrays with bounds that are nonconstant expressions are of two kinds:

- An *adjustable array* is an explicit-shape *dummy* array with a bound that depends on the values of one or more variables.
- An *automatic array* is an explicit-shape *local* array with a bound that depends on the values of one or more variables. An automatic array is always a CM array.

Neither an adjustable array nor an automatic array may appear in a `SAVE` or `DATA` statement, or be declared with a `SAVE` or `DATA` attribute.

The bounds, and hence the shape, of an adjustable array or an automatic array are fixed at each entry to the defining procedure. The bounds of such an array are unaffected by any redefinition of the variables referenced in the specification expression.

Examples of explicit-shape array specifiers:

```
( N:N+99 )
( 10, 100, N*N )
( -1000:2000, N+1 )
( 0:N-1 )
( N, N, N, N, N, N, N )
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The examples above are valid if N is the name of a constant, a dummy variable, or a variable in common. (N cannot be the name of a variable declared local to a procedure or to the main program.)

5.2.2 Assumed-Shape Arrays

An *assumed-shape* array is a dummy array (see Chapter 12) without explicitly declared upper bounds that assumes the shape of an actual argument array passed to the procedure in which it is declared. An assumed-shape array is declared by using the name of a dummy argument in a type declaration or DIMENSION statement that contains an *assumed-shape array specifier* (R5-6). An assumed-shape array is always a CM array.

An *assumed-shape-array-spec* is: R5-6
assumed-shape-spec [, *assumed-shape-spec*]...

An *assumed-shape-spec* is: R5-7
[*lower-bound*] :

A *lower-bound* is: R5-4
specification-expr

- The maximum number of *assumed-shape-spec*s in an *assumed-shape-array-spec* is 31.

The rank of an assumed-shape array is equal to the number of colons in the *assumed shape array specification* used in its declaration.

The size of a particular dimension of an assumed-shape array is the size of the corresponding dimension of the associated actual argument array. If the lower bound value is d and the size of the corresponding dimension of the associated actual argument array is s, then the value of the declared upper bound is $s + d - 1$. The declared lower bound is *lower-bound*, if present, and 1 otherwise. The lower bound must be a specification expression, as with explicit shape arrays.

Examples of assumed-shape array specifiers:

```
( N: )
( 10:, 100:, N*N: )
( -1000:, N+1: )
( 0:, 0: )
( :, :, :, :, :, :, : )
```

The examples above are valid if N is the name of a constant, a dummy variable, or a variable in common.

Copyright © 1989–1994 Thinking Machines Corporation

The following example illustrates the use of assumed-shape arrays. The subroutine INIT2INT initializes an integer array of rank two, while subroutine PRINT2INT prints an integer array of rank two.

```
      SUBROUTINE INIT2INT( A )
      INTEGER A(:,:)
      INTEGER U(2)
      U = UBOUND( A )
      FORALL( I=1:U(1), J=1:U(2) )
     $  A(I,J) = I*10 + J
      END

      SUBROUTINE PRINT2INT( A )
      INTEGER A(:,:)
C     ... ! code to print array A
      END

      PROGRAM TEST
      INTEGER A(4,7), B(2,9)
      CALL INIT2INT( A )
      CALL PRINT2INT( A )
      END
```

5.2.3 Deferred-Shape Arrays

A *deferred-shape* array is an ALLOCATABLE array or array pointer whose rank is specified but whose extents are not determined until space is dynamically allocated for the array by the execution of an ALLOCATE statement. These arrays can also be deallocated during program execution by calling the DEALLOCATE statement. Deferred-shape arrays are especially useful if the extents of the array are to be input or calculated and so are not known until run time. ALLOCATE and DEALLOCATE also allow the programmer to manage the use of memory within the application; arrays that are used only within a program unit can be allocated within the unit and then deallocated before leaving the unit.

An array is given the ALLOCATABLE or POINTER attribute in a type declaration statement or in an ALLOCATABLE or POINTER statement using a *deferred-shape-array-spec*.

A *deferred-shape-array-spec* is: R5-8

```
: [ , : ]...
```

The rank of the array is equal to the number of colons in the *deferred-shape-array-spec*. Examples of deferred-shape array specifications are:

```
REAL,ALLOCATABLE :: D(:,:)
INTEGER,POINTER :: R(:,:)
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

More information on the ALLOCATABLE attribute is given in Section 5.4.6. More information on the ALLOCATABLE statement is given in Section 5.5.5. The POINTER attribute is described in Section 5.4.7, and information on the POINTER statement is given in Section 5.5.6. The ALLOCATE statement is described in Section 6.3. The ALLOCATED intrinsic function is described in Section 17.17.

5.2.4 Assumed-Size Arrays

An *assumed-size* array is a dummy array (see Chapter 12) whose size in the last dimension is unknown, and must be assumed from other information provided to the procedure, either from other dummy arguments or through common variables. The rank and extents may differ for the actual and dummy array, but the size of the actual array must be at least the size assumed by the declaring procedure for the dummy array. An assumed-size array is always a front-end array.

An *assumed-size-array-spec* is: R5-9
*[explicit-shape-spec ,]... [lower-bound :] **

An *explicit-shape-spec* is: R5-3
[lower-bound :] upper-bound

A *lower-bound* is: R5-4
specification-expr

An *upper-bound* is: R5-5
specification-expr

- The maximum number of *explicit-shape-spec*s in an *assumed-size-array-spec* is 30.
- The value to be returned by an array-valued function must not be declared as an assumed-size array.

The size of an assumed-size dummy array is determined as follows:

- If the actual argument associated with the assumed-size dummy array is an array of any type other than character, the size is that of the actual array.
- If the actual argument associated with the assumed-size dummy array is an array element of any type other than character, with a subscript order value of *r* in an array of size *x*, the size of the dummy array is MAX(*x*−*r*+1, 0).

Copyright © 1989–1994 Thinking Machines Corporation

- If the actual argument is a character array, a character array element, or a substring of a character array element, and if it begins at character storage unit `t` of an array with `c` character storage units, the size of the dummy array is

  ```
  MAX( INT((c-t+1)/e), 0 )
  ```

  where `e` is the length of an element in the dummy character array.

The rank of an assumed-size array is one greater than the number of *explicit-shape-spec*s in its specification. If an assumed-size array has rank $n > 1$, the product of the extents of the first $n - 1$ dimensions must be less than or equal to the size of the associated actual array.

An assumed-size array has no upper bound in its last dimension and therefore has no extent in its last dimension and no shape.

The declared bounds of the first $n - 1$ dimensions are those specified by the *explicit-shape-spec*s (R5-3), if present, in the *assumed-size-array-spec* (R5-9). The declared lower bound of the last dimension is *lower-bound* (R5-4), if present, and 1 otherwise. An assumed-size array may be subscripted or sectioned. If a section subscript list is provided in which a subscript or section with an upper bound for the last dimension is present, the resulting array has a declared shape and size.

If an assumed-size array has bounds that are nonconstant specification expressions, the bounds are fixed at each entry to the procedure. The bounds of such an array are unaffected by any redefinition or undefinition during execution of the procedure of any variables referenced by the specification expressions defining the array bounds.

An assumed-size array is permitted to appear as a whole array in an executable construct, but only as an actual argument in external procedure references. (See Section 12.5.)

Examples of assumed-size array specifiers:

```
( N:N+99, 1:* )
( 10, 100, N*N, 10:* )
( -1000:2000, N+1, * )
( 0:N-1, 0:* )
( N, N, N, N, N, N, * )
```

The examples above are valid if `N` is the name of a constant, a dummy variable, or a variable in common.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

5.3 Type Declaration Statements

A *type declaration statement* (R5-10) declares one or more objects or functions to have a given type and may specify other attributes of the entity, including an initial value.

A *type-decl-stmt* is one of: R5-10

- *type-spec entity-decl(s)*
- *type-spec [, attr-spec]...* `::` *entity-decl(s)*

A *type-spec* is one of: R5-11

- `INTEGER` *[kind-selector]*
- `REAL` *[kind-selector]*
- `DOUBLE PRECISION`
- `COMPLEX` *[kind-selector]*
- `DOUBLE COMPLEX`
- `CHARACTER` *[length-spec]*
- `LOGICAL`

A *kind-selector* is: R5-12

([`KIND=`] *scalar-int-initialization-expr)*

An *attr-spec* is one of: R5-13

- `DIMENSION` (*array-spec*)
- `PARAMETER`
- `SAVE`
- `DATA`
- `INTENT` (*intent-spec*)
- `ALLOCATABLE`
- `POINTER`

NOTE: CM Fortran also accepts the attribute `ARRAY`; `ARRAY` is an earlier name for the `DIMENSION` attribute.

An *entity-decl* is one of: R5-14

- object-*name [* (*array-spec*) *] [* `*` *char-length] [* = constant-*expr]*
- function-*name [* `*` *char-length]*

- The value of a *scalar-int-initialization-expr* must be nonnegative and must specify a representation method that exists on the processor.

- Syntactic restrictions are given in the following sections for the numeric, logical, and character forms of type declaration statements, described in rules (R5-18), (R5-21), and (R5-23), respectively.

Both forms of type declaration statement (R5-10) begin with a *type specifier* (R5-11) to indicate the type of all objects and functions declared by the statement. This type may over-

Copyright © 1989–1994 Thinking Machines Corporation

ride or confirm the type implicitly assumed based on the first letter of the object or function name. In the example

```
REAL A1, A2
INTEGER A3
```

the declarations of A1 and A2 confirm their implicitly assumed types, and the declaration of A3 overrides its assumed type. Section 5.6 describes the implicit typing rules assumed in a program unit or interface block, and how the IMPLICIT statement may be used to change those rules.

The second form of type declaration statement may include a list of *attribute specifiers* (R5-13) to further characterize the nature of the objects or functions being declared. The use of attribute specifiers in type declaration statements is described in Section 5.4.

The supported scalar integer constant expressions for indicating KIND in the kind selector are listed in the following table:

Scalar Integer Initialization Expression	Data Type	Length
_SINGLE_INT_	INTEGER (32-bit)	32
_DOUBLE_INT_	INTEGER (64-bit)*	64
SINGLE	REAL	32
DOUBLE	DOUBLE	64
_SINGLE_COMPLEX_	COMPLEX	32
_DOUBLE_COMPLEX_	DOUBLE COMPLEX	64

NOTE: The reason _DOUBLE_COMPLEX_ is 64 is so that double precision and double complex have the same value, allowing statements such as

```
Z = CMPLX( 1D0, 2D0, KIND(0D0) )
```

Each *entity declarator* (R5-14) names an object or function being declared, and may specify additional attributes of that particular entity. Any attributes specified in the declarator override similar attributes specified in the attribute specifier. An optional constant expression in a declarator may be provided to supply an object with an initial value. Attribute specifiers and entity declarators are described in Section 5.4 for the seven different attribute forms.

A name that identifies a specific intrinsic function in a program unit or interface block has a type as specified in Section 17.16. An explicit type declaration statement is not required to reference such a name, but is permitted. The appearance of a generic function name in a type declaration statement is not sufficient by itself to override the generic properties of

* The 64-bit integer type is supported only on CM-5 systems with vector units.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

that function. (Its name must also appear in an EXTERNAL statement, described in Section 12.7.)

Examples of the first form of type declaration statement are given in the following sections. Examples of type declaration statements using attribute specifiers are given in Sections 5.4.1 through 5.4.6.

Type Declaration Statements — Alternative Forms

An alternative form of type declaration statement is available for compatibility with a widely accepted Fortran 77 extension. In this form, a *type specifier* (R5-15) of INTEGER, REAL, COMPLEX, or LOGICAL is followed by a *byte length* (R5-17) to specify a default precision (in terms of bytes) for objects being declared in the *entity-decl(s)* list. Within a numeric or logical type declaration statement, an individual *entity declarator* (R5-16) may itself be qualified by a byte length specifier to override the default precision for that object.

A *type-decl-stmt* is one of: R5-10
- *type-spec entity-decl(s)*
- *type-spec [, attr-spec]...* **::** *entity-decl(s)*

A *type-spec* is one of: R5-15
- INTEGER *[* byte-length]*
- REAL *[* byte-length]*
- COMPLEX *[* byte-length]*
- LOGICAL *[* byte-length]*

An *entity-decl* is one of: R5-16
- object-*name [* byte-length] [* (*array-spec*) *] [* = constant-*expr]*
- object-*name [* (*array-spec*) *] [* = constant-*expr]*
- function-*name*

A *byte-length* is: R5-17
integer-literal-constant

Only byte-length specifiers of 4, 8, and 16 are valid. The table below shows the valid type specifiers of this form and its equivalent form from Section 5.3.

Copyright © 1989–1994 Thinking Machines Corporation

```
INTEGER*4    ↔    INTEGER
INTEGER*8    ↔    INTEGER
REAL*4       ↔    REAL
REAL*8       ↔    DOUBLE PRECISION
COMPLEX*8    ↔    COMPLEX
COMPLEX*16   ↔    DOUBLE COMPLEX
LOGICAL*4    ↔    LOGICAL
```

Examples of type declaration statements using byte-length specifiers are given in the next two sections.

5.3.1 Numeric Type Declaration Statements

The numeric type declaration statement (R5-18) declares one or more objects or functions to be of a numeric type. This type declaration statement is a form of the general type declaration statement (R5-10).

A (numeric) *type-decl-stmt* is one of: R5-18

- *num-type-spec num-entity-decl(s)*
- *num-type-spec [* , *attr-spec]...* **::** *num-entity-decl(s)*

A *num-type-spec* is one of: R5-19

- `INTEGER` *[* `*` *byte-length]*
- `INTEGER` *[* `(KIND=` *scalar-int-initialization-expr*`)` *]*
- `REAL` *[* `*` *byte-length]*
- `REAL` *[* `(KIND=` *scalar-int-initialization-expr*`)` *]*
- `DOUBLE PRECISION`
- `COMPLEX` *[* `*` *byte-length]*
- `COMPLEX` *[* `(KIND=` *scalar-int-initialization-expr*`)` *]*
- `DOUBLE COMPLEX`

A *num-entity-decl* is one of: R5-20

- object-*name [* `*` *byte-length] [* `(` *array-spec* `)` *] [* = constant-*num-expr]*
- function-*name*

- The value of a *scalar-int-initialization-expr* must be nonnegative and must specify a representation method that exists on the processor (32 or 64).
- The optional constant-*num-expr* in a *num-entity-decl* must appear if and only if the second form of *num-type-decl-stmt* is used and it specifies a `PARAMETER` or `DATA` attribute.
- A given *attr-spec* may not appear more than once in a single *type-decl-stmt*.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- The PARAMETER, DATA, and SAVE attributes must not be specified for dummy arguments, functions, or objects in a common block. However, the attribute DATA may be specified using a DATA statement for objects in a named common block, provided the DATA statement appears in a block data program unit.

An integer data item is an exact representation of an integer number. It may assume only integral values that are positive, negative, or zero. An INTEGER*4 variable occupies a single numeric storage unit in a storage sequence. An INTEGER*8 variable occupies two numeric storage units in a storage sequence.

A real data item approximates the value of a real number. It may assume a positive, negative, or zero value. A single-precision real variable occupies a single numeric storage unit in a storage sequence. A double-precision real variable occupies two numeric storage units in a storage sequence.

A complex data item approximates the value of a complex number. It may assume a zero value or a nonzero value. A single-precision complex variable occupies two numeric storage units in a storage sequence; the real part occupies the first storage unit and the imaginary part occupies the second storage unit. A double-precision complex variable occupies four numeric storage units in a storage sequence; the real part occupies the first two storage units and the imaginary part occupies the last two storage units.

Examples of numeric type declaration statements without attribute specifier lists:

```
INTEGER I, J, K         ! scalar integer variables
INTEGER*4 II, JJ, KK    ! ditto
INTEGER*8 III, JJJ, KKK ! 64-bit integer variables
INTEGER(KIND=64) A      ! ditto
INTEGER(KIND=_DOUBLE_INT_) B        ! 64-bit integer
REAL(KIND=_DOUBLE_) C   ! double-precision real
REAL*8 A(100,100)       ! double-precision real array of rank 2
REAL*4 X(10), Y*8(10)   ! X is single-, Y is double-precision;
                        ! both are arrays
REAL A(10)  ! explicit-shape real array of rank 1, size 10
DOUBLE PRECISION B(10)  ! same, but double-precision
REAL A(-5:5,0:N-1)   ! explicit-shape real array of rank 2;
                     ! size is 11*N, indexes start at -5 and 0
COMPLEX MATRIX(5,5,5)   ! a complex 3-D cube
DOUBLE COMPLEX F, G, H
COMPLEX(KIND=_SINGLE_COMPLEX_) M    ! single-precision complex
COMPLEX*16 N            ! double-precision complex
```


Copyright © 1989–1994 Thinking Machines Corporation

```
SUBROUTINE SUB( N, A, X, Y, S )
REAL A(N,10)      ! explicit-shape (adjustable), rank 2
REAL L(N,10)      ! explicit-shape (automatic), rank 2
REAL X(:), Y(0:)  ! assumed-shape real arrays of rank 1
REAL S(N,*)       ! assumed-size real array of rank 2
...
INTEGER LL
PARAMETER (LL=64)
INTEGER(KIND=LL) D   ! 64-bit integer
REAL(KIND=LL) E      ! double-precision real
```

5.3.2 Typed Constants

To specify the type (including kind) of a literal constant, append an underscore and then an integer constant expression that evaluates to a valid `KIND` number (32 or 64). For example, a constant of type `REAL(KIND=_DOUBLE_)` could be written either as `1.0D0` or as `1.0__DOUBLE_`.

Note the double underscore when the appended underscore of the typed constant is combined with the prepended underscore of the CM Fortran symbolic name for the kind type parameter:

```
INTEGER LL
PARAMETER (LL=64)
INTEGER(KIND=LL) D
D = 1234_64 + 1234_LL + 1234__DOUBLE_INT_
```

A typed constant provides a way to coerce variables to the desired length. (An alternative is the intrinsic functions `KIND`, `INT`, and `NINT`, described in Chapter 17.) For example, to coerce an `INTEGER*4` variable N to type `INTEGER*8`, use:

```
N = N + 0_64
```

5.3.3 LOGICAL Type Declaration Statement

The `LOGICAL` type declaration statement (R5-21) declares one or more objects or functions to be of type logical. This type declaration statement is a form of the general type declaration statement (R5-10).

A (LOGICAL) *type-decl-stmt* is one of: R5-21

- `LOGICAL` *[* byte-length] logical-entity-decl(s)*
- `LOGICAL` *[* byte-length] [, attr-spec]...* **::** *logical-entity-decl(s)*

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

A *logical-entity-decl* is one of: R5-22

- object-*name [* byte-length] [(array-spec)] [* = logical-constant-*expr]*
- function-*name*

- The optional logical-constant-*expr* in a *logical-entity-decl* must appear if and only if the second form of *logical-type-decl-stmt* is used and it specifies a `PARAMETER` or `DATA` attribute.
- A given *attr-spec* may not appear more than once in a single *type-decl-stmt*.
- The `PARAMETER`, `DATA`, and `SAVE` attributes must not be specified for dummy arguments, functions, or objects in a common block. However, the attribute `DATA` may be specified using a `DATA` statement for objects in a named common block, provided the `DATA` statement appears in a block data program unit.

A logical data item may assume only the values true or false, represented by the literal values `.TRUE.` and `.FALSE.`, respectively. A logical data item occupies a single numeric storage unit in a front-end storage sequence. (An element of a logical CM array occupies 32 bits.)

Examples of `LOGICAL` type declaration statements without attribute specifier lists:

```
LOGICAL DONE, EOF_SEEN

LOGICAL MASK(3,2)

LOGICAL MASK1(5,3), MASK2(5,3) ! logical arrays, shape [5,3]

LOGICAL*4 REPEAT     ! logical scalar
```

5.3.4 CHARACTER Type Declaration Statement

The `CHARACTER` type declaration statement (R5-23) declares one or more objects or functions to be of type character. The statement may specify a default length for all character objects and functions declared. This type declaration statement is a form of the general type declaration statement (R5-10).

A (CHARACTER) *type-decl-stmt* is one of: R5-23

- `CHARACTER` *[* default-length [,]] char-entity-decl(s)*
- `CHARACTER` *[* default-length] [, attr-spec]... :: char-entity-decl(s)*

A *char-entity-decl* is one of: R5-24

- object-*name [(array-spec)] [* char-length] [* = constant-*char-expr]*
- function-*name [* char-length]*

Copyright © 1989–1994 Thinking Machines Corporation

A *default-length* is: R5-25
char-length

A *char-length* is one of: R5-26
- *integer-literal-constant*
- (scalar-integer-constant-*expr*)
- (*)

- The optional constant-*char-expr* in a *char-entity-decl* must appear if and only if the second form of CHARACTER *type-decl-stmt* is used and it specifies a PARAMETER or DATA attribute.
- A given *attr-spec* may not appear more than once in a single *type-decl-stmt*.
- The PARAMETER, DATA, and SAVE attributes must not be specified for dummy arguments, functions, or objects in a common block. However, the attribute DATA may be specified using a DATA statement for objects in a named common block, provided the DATA statement appears in a block data program unit.

The character length specified as the *default-length* (R5-25) is the default length of the character objects or functions being declared. If a *char-length* (R5-26) is specified for a given entity, it specifies the length of that single object or function and overrides the default length established by the *default-length*. If neither a *default-length* nor a *char-length* is specified, the length of the character entity is 1. If a *char-length* is not an asterisk, it must evaluate to a positive value.

A *char-length* of * may be used in the following ways:

- In the declaration of a named constant, a length attribute value of * indicates that the length of the named constant is the length of the specified character constant expression.
- In the declaration of the dummy argument of a procedure, a length attribute value of * indicates that the dummy argument is to assume the length of the associated actual argument when the procedure is invoked. Such a dummy argument is called an *assumed-length* character string.
- Within the specification part of an external function or a function interface block, a length attribute value of * for the function result indicates that the length of the result will be specified (possibly differently) by each program unit that references the function. The length of the function result is specified in an EXTERNAL statement or an interface block in each program unit that references the function.

The length specified for a character-valued statement function or for a statement function dummy argument of type character must be an integer constant expression.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

A scalar object of type character has a storage sequence of character storage units. The number of character storage units in the storage sequence is the length of the character entity. The order of the sequence corresponds to the ordering of character positions.

Examples of CHARACTER type declaration statements without attribute specifier lists:

```
CHARACTER*10  A, B      ! scalar character strings of length 10
CHARACTER*(8+2) C, D    !  ditto

CHARACTER*10 B, C*20    ! B has length 10, C has length 20

CHARACTER*10 BB(12)     ! An array of 12 strings,
                        !  each of length 10
```

The following function removes the first occurrence of a substring from a string, and returns the remainder of the string as an assumed-length result; the original string is returned if the substring is not found.

```
      CHARACTER*(*) FUNCTION REMOVE( SUBSTR, STRING )
      CHARACTER*(*) SUBSTR, STRING
      N = INDEX( STRING, SUBSTR )
      IF ( N .NE. 0 ) THEN
        REMOVE = STRING( 1:N-1 )
     &           // STRING( N+LEN(SUBSTR):LEN(STRING) )
      ELSE
        REMOVE = STRING
      END IF
      END
```

A reference to the function REMOVE must be preceded by a declaration to establish the length of the result:

```
      ...
      CHARACTER*80 REMOVE, RESULT, STRING
      STRING = 'QUICK BROWN FOX'
      RESULT = REMOVE( 'BROWN', STRING )
```

5.3.4.1 CHARACTER Type Declaration Statement — Alternative Forms

The CM Fortran language also includes the following syntactic forms of CHARACTER type declaration statements, as alternatives to those defined in Section 5.3.4. (The forms are provided for compatibility with Fortran 90 syntax.)

Copyright © 1989–1994 Thinking Machines Corporation

A (CHARACTER) *type-decl-stmt* is one of: R5-27

- `CHARACTER ( [ LEN = ] * )` *char-entity-decl(s)*
- `CHARACTER ( [ LEN = ]` *default-len* `) [,]` *char-entity-decl(s)*
- `CHARACTER ( [ LEN = ] * ) [,` *attr-spec* `]... ::` *char-entity-decl(s)*
- `CHARACTER ( [ LEN = ]` *default-len* `) [,` *attr-spec* `]... ::` *char-entity-decl(s)*

A *default-len* is: R5-28

scalar-int-const-*expr*

The relationship of the above forms to those described in Section 5.3.4 is as follows:

`CHARACTER ( [ LEN = ] * )` *char-entity-decl(s)* →
`CHARACTER*(*) [,]` *char-entity-decl(s)*

`CHARACTER ( [ LEN = ]` *default-len* `) [,]` *char-entity-decl(s)* →
`CHARACTER*(`*default-len*`) [,]` *char-entity-decl(s)*

`CHARACTER ( [ LEN = ] * ) [,` *attr-spec* `]... ::` *char-entity-decl(s)* →
`CHARACTER*(*) [,` *attr-spec* `]... ::` *char-entity-decl(s)*

`CHARACTER ( [ LEN = ]` *default-len* `) [,` *attr-spec* `]... ::` *char-entity-decl(s)* →
`CHARACTER*(`*default-len*`) [,` *attr-spec* `]... ::` *char-entity-decl(s)*

The following are valid examples of character declarations of these alternative forms; the comments indicate the equivalent Fortran 77 forms.

```
CHARACTER(LEN=5) E            ! CHARACTER*(5) E
CHARACTER(5) F                ! CHARACTER*(5) F
CHARACTER(5+3) G              ! CHARACTER*(5+3) G
CHARACTER(LEN=5+3) H          ! CHARACTER*(5+3) H
CHARACTER(*) I                ! CHARACTER*(*) I
CHARACTER(LEN=*) J            ! CHARACTER*(*) J
```

5.4 Attribute Specification in Type Declaration Statements

Attribute specifiers may appear in the attributed form of type declaration statement (the second alternative of R5-10, page 39) to provide additional information regarding the objects or functions being declared, or to specify restrictions on the use of an object. These *attributed forms* of type declaration statements (R5-29) are described in the following sections.

An (attributed) *type-decl-stmt* is: R5-29

type-spec `[ ,` *attr-spec* `]... ::` *entity-decl(s)*

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

An *attr-spec* is one of: R5-13

- DIMENSION (*array-spec*)
- PARAMETER
- SAVE
- DATA
- INTENT (*intent-spec*)
- ALLOCATABLE
- POINTER

NOTE: CM Fortran also accepts the attribute ARRAY; ARRAY is an earlier name for the DIMENSION attribute.

An *entity-decl* is one of: R5-14

- object-*name [* (*array-spec*) *] [* = constant-*expr]*
- function-*name*

- The second form of *entity-decl* (naming a function) must not be used in a type declaration statement that specifies a PARAMETER, SAVE, DATA, or POINTER attribute.
- The PARAMETER, SAVE, and DATA attributes must not be specified for dummy arguments, function results, adjustable arrays, automatic arrays, or for objects in a common block. However, the DATA attribute may be specified using a DATA statement for objects in a named common block provided the DATA statement appears in a block data program unit.
- The INTENT attribute may be specified only for dummy arguments (see Chapter 12).

Examples of the attributed form of type declaration statement:

```
INTEGER :: I, J
INTEGER, DATA :: COUNT = 0

INTEGER, DIMENSION(3), DATA :: ORDER = [1,2,3]

LOGICAL :: DONE, EOF_SEEN

REAL, DATA :: ONE = 1.0, PI = 22.0 / 7.0

COMPLEX, PARAMETER :: ORIGIN = (0,0)

CHARACTER*10 :: B, C*20    ! B has length 10, C has length 20

CHARACTER*10 :: X, Y ! scalar character strings of length 10
CHARACTER*(8+2), DATA :: C='1234567890' ! ditto, initialized

INTEGER, PARAMETER :: N = 8
CHARACTER*(N) :: C(12)  ! An array of 12 strings of length 8

CHARACTER*(*), PARAMETER :: LIB = '/usr/lib'  ! named constant
```


Copyright © 1989–1994 Thinking Machines Corporation

```
FUNCTION NOTGT( C1, C2 )
CHARACTER*(*) :: C1, C2    ! assumed-length character strings
NOTGT = C1 .LE. C2
END
```

The attribute specifiers are defined in the following sections.

5.4.1 DIMENSION Attribute

A type declaration statement containing a DIMENSION attribute declares the entities named in the statement to be arrays. This form of type declaration statement (R5-30) is a special case of the attributed form of type declaration statement (R5-29).

A (DIMENSION-attributed) *type-spec* is: R5-30
type-spec, *dimension-attr-spec-list* :: *entity-decl(s)*

A *dimension-attr-spec-list* is: R5-31
[save-or-data-spec(s),] [DIMENSION (*array-spec*) *] [, save-or-data-spec(s)]*

A *save-or-data-spec* is one of: R5-32
- SAVE
- DATA

An *entity-decl* is one of: R5-14
- object-*name [* (*array-spec*) *] [* = constant-*expr]*
- function-*name*

NOTE: CM Fortran also accepts the attribute ARRAY; ARRAY is an earlier name for the DIMENSION attribute.

The rank and shape of each array entity is specified by the *array-spec* in the *entity-decl* if there is one, or by the *array-spec* in the DIMENSION attribute otherwise. An *array-spec* in an *entity-decl* specifies the rank of a single array, and overrides the *array-spec* in the DIMENSION attribute. If the entity being declared is not an assumed-size or assumed-shape dummy argument, the DIMENSION attribute also specifies its shape. If the DIMENSION attribute is omitted, an *array-spec* must be specified in the *entity-decl* to declare an array.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The declaration statement

```
REAL, DIMENSION(-5:+5) :: X, Y, Z
```

declares three objects with names X, Y, and Z, all of which are explicit-shape, rank-one arrays of type single-precision real with a lower bound of –5, an upper bound of +5, and (consequently) a size of 11.

Other examples of numeric type declaration statements specifying a DIMENSION attribute:

```
! explicit-shape real vector of size 10, saved and initialized
REAL, DIMENSION(10), SAVE, DATA :: A = [1:10]

INTEGER, DIMENSION(10,10), SAVE :: A,B,C ! 3 2D integer arrays

REAL, DIMENSION(10) :: X, Y, Z(5,5) !2 1D and 1 2D real array

SUBROUTINE SUB( N, A, L, X, Y, S )
REAL, DIMENSION(N,10) :: A! explicit-shape(adjustable), rank 2
REAL, DIMENSION(N,10) :: L! explicit-shape (automatic), rank 2
REAL, DIMENSION(:) :: X   ! assumed-shape real array, rank 1
REAL, DIMENSION(0:) :: Y  ! assumed-shape real array, rank 1
REAL, DIMENSION(N,*) :: S ! assumed-size real array, rank 2
...

REAL, DIMENSION(-5:5,0:N-1) :: A, B, C ! explicit-shape real
    ! arrays of rank 2; size is 11*N, indexes start at -5 and 0

LOGICAL, DIMENSION(5,3) :: MASK1, MASK2 ! logical, shape [5,3]
```

5.4.2 PARAMETER Attribute

A type declaration statement specifying a PARAMETER attribute declares the objects named in the statement to be named scalar constants and provides their values. This form (R5-33) of type declaration statement is a special case of the attributed form of type declaration statement (R5-29).

A (PARAMETER-attributed) *type-decl-stmt* is: R5-33
type-spec, *parameter-attr-spec* :: *entity-decl(s)*

A *parameter-attr-spec* is: R5-34
PARAMETER

A *parameter-attr-entity-decl* is: R5-35
object-*name* = constant-*expr*

Copyright © 1989–1994 Thinking Machines Corporation

- The PARAMETER attribute must not be specified for dummy arguments, function results, adjustable arrays, automatic arrays, or for objects in a common block.

The appearance of a PARAMETER attribute specifier in a type declaration statement requires that a constant-*expr* be specified in each *entity-decl* to provide a value for the declared object. The object-*name* is defined with the value determined by the constant-*expr*, in accordance with the rules of assignment conformance (Section 10.1.1). Any named constant that appears in the constant-*expr* must have been defined previously.

Examples of type declaration statements specifying the PARAMETER attribute:

```
REAL, PARAMETER :: ONE = 1.0, PI = 22.0 / 7.0

INTEGER, PARAMETER :: ORDER = 2

LOGICAL, PARAMETER :: T = .TRUE., F = .FALSE.

CHARACTER*(*), PARAMETER :: PATH = "/usr/bin/temp"
      ! PATH is a named constant with length equal to that
      !     of the literal string
```

5.4.3 DATA Attribute

A type declaration statement specifying a DATA attribute declares one or more variables and provides them with initial values. This form (R5-36) of type declaration statement is a special case of the attributed form of type declaration statement (R5-29).

A (DATA-attributed) *type-decl-stmt* is: R5-36
type-spec, *data-attr-spec-list* :: *entity-decl(s)*

A *data-attr-spec-list* is: R5-37
[save-or-dimension-spec(s) ,] DATA *[, save-or-dimension-spec(s)]*

A *save-or-dimension-spec* is one of: R5-38
- SAVE
- DIMENSION (*array-spec*)

A *data-attr-entity-decl* is: R5-39
object-*name [* (*array-spec*) *]* = constant-*expr*

- The DATA attribute must not be specified for dummy arguments, function results, adjustable arrays, automatic arrays, or for objects in a common block. However, the DATA attribute may be specified using a DATA statement for objects in a named common block provided the DATA statement appears in a block data program unit.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The appearance of a DATA attribute specifier in a type declaration statement requires that a constant-*expr* be specified in each *entity-decl* to provide an initial value for the declared object. The object-*name* is defined with the value determined by the constant-*expr*, in accordance with the rules of assignment conformance (Section 10.1.1). Any named constant that appears in the constant expression defining an initial value must have been defined previously.

The presence of a DATA attribute in a type declaration statement implies that all the objects declared in the statement are saved, except for objects in a named common block. The implied SAVE attribute may be reaffirmed by explicit use of the SAVE attribute in a type declaration statement, or by the inclusion of the object names in a SAVE statement.

Examples of type declaration statements specifying DATA attributes (as well as other attributes):

```
INTEGER, DIMENSION(3), DATA :: ORDER = [ 1, 2, 3 ]

REAL, DATA :: ONE = 1.0, PI = 22.0 / 7.0

REAL, DIMENSION(3), DATA :: D = [ 1.0, 2.0, 3.0 ]

COMPLEX, DATA :: CUBE_ROOT = ( -0.5, 0.866 ) ! a complex scalar
                        ! variable with an initial value

LOGICAL, DATA :: DONE = .FALSE., EOF_SEEN = .FALSE.

LOGICAL, DIMENSION(3), DATA :: MASK=[.FALSE., .FALSE., .TRUE.]

LOGICAL, DATA :: DONE = .FALSE., EOF_SEEN = .FALSE.

CHARACTER*10, DATA :: AA = 'xyzzy'
                ! AA has length 10, is blank-filled on right

CHARACTER*(*), DATA :: PATH = "/usr/bin/temp"
      ! PATH is a variable with length equal to that
      !     of the literal string
```

5.4.4 SAVE Attribute

A type declaration statement specifying a SAVE attribute indicates that the objects named in the statement are to retain their definition and value after execution of a RETURN or END statement in the program unit containing the declaration. This form (R5-40) of type declaration statement is a special case of the attributed form of type declaration statement (R5-29).

A (SAVE-attributed) *type-decl-stmt* is: R5-40
type-spec, *save-attr-spec-list* :: *save-attr-entity-decl(s)*

Copyright © 1989–1994 Thinking Machines Corporation

A *save-attr-spec-list* is: R5-41
[data-or-array-spec(s) ,] SAVE *[, data-or-array-spec(s)]*

A *data-or-array-spec* is one of: R5-42
- DATA
- DIMENSION (*array-spec*)

A *save-attr-entity-decl* is: R5-43
object-*name [* (*array-spec*) *] [* = constant-*expr]*

- The SAVE attribute must not be specified for dummy arguments, function results, adjustable arrays, automatic arrays, or for objects in a common block.

An object declared with the SAVE attribute is called a *saved object*. The SAVE attribute may appear in declarations in a main program, but has no effect (since all objects declared in the main program exist for the duration of program execution).

Examples of type declaration statements specifying the SAVE attribute (as well as other attributes):

```
INTEGER, DIMENSION(3), SAVE, DATA :: ORDER = [1,2,3]

REAL, DIMENSION(3), SAVE, DATA :: D = [1.0,2.0,3.0]

LOGICAL, SAVE :: T, F

LOGICAL, SAVE, DATA :: DONE = .FALSE., EOF_SEEN = .FALSE.

CHARACTER*(*), SAVE, DATA :: PATH = "/usr/bin/temp"
      ! PATH is a named constant with length equal to that
      !     of the literal string (13)
```

5.4.5 INTENT Attribute

The INTENT attribute specifies the intended use of a dummy argument by a procedure. (Procedures and dummy arguments are discussed in Chapter 12.)

An (INTENT-attributed) *type-decl-stmt* is: R5-44
type-spec, *intent-attr-spec-list* :: *intent-attr-entity-decl(s)*

An *intent-attr-spec-list* is one of: R5-45
- INTENT (*intent-spec*)
- DIMENSION (*array-spec*) , INTENT (*intent-spec*)
- INTENT (*intent-spec*) , DIMENSION (*array-spec*)

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

An *intent-spec* is one of: R5-46

- `IN`
- `OUT`
- `INOUT`

An *intent-attr-entity-decl* is: R5-47

object-*name [* `(` *array-spec* `)` *]*

- The `INTENT` attribute may appear only in a type declaration statement in the specification part of a subprogram or interface block.
- The `INTENT` attribute must not be specified for a dummy argument that is a dummy procedure.

The `INTENT(IN)` attribute specifier indicates that a dummy argument named in an *intent-attr-entity-decl(s)* list has intent `IN`. A dummy argument with intent `IN` must not be changed during the execution of the procedure.

The `INTENT(OUT)` attribute specifier indicates that a dummy argument named in an *intent-attr-entity-decl(s)* list has intent `OUT`. A dummy argument with intent `OUT` must be defined within the declaring procedure before a reference to it is made, and any actual argument associated with the dummy argument must be definable. A dummy argument with intent `OUT` is undefined when a procedure is invoked (but may be defined by the procedure).

The `INTENT(INOUT)` attribute specifier indicates that a dummy argument named in an *intent-attr-entity-decl(s)* list has intent `INOUT`. A dummy argument with intent `INOUT` is intended for use both to receive data and to return data to the invoking procedure or program. Any actual argument that becomes associated with such a dummy argument must be definable.

If no `INTENT` attribute is specified for a dummy argument, its use is subject to the limitations of the associated actual argument.

The example below is a subroutine containing type declaration statements specifying an `INTENT` attribute for each of its three dummy arguments:

```
SUBROUTINE MOVE( FROM, TO, N )
INTEGER, DIMENSION(N), INTENT(IN) :: FROM
INTEGER, DIMENSION(N), INTENT(OUT) :: TO
INTEGER, INTENT(IN) :: N
...
```


Copyright © 1989–1994 Thinking Machines Corporation

5.4.6 ALLOCATABLE Attribute

A type declaration statement specifying an ALLOCATABLE attribute declares the objects named in the statement to be allocatable, *deferred-shape* arrays (see Section 5.2.3). This form (R5-48) of the type declaration statement is a special case of the attributed form of type declaration statement (R5-29).

An (ALLOCATABLE-attributed) *type-decl-stmt* is: R5-48
type-spec, *allocatable-attr-spec* :: *entity-decl(deferred-shape-array-spec)*

An *allocatable-attr-spec* is: R5-49
ALLOCATABLE

A *deferred-shape-array-spec* is: R5-8
: [, :]...

The rank of the array is equal to the number of colons in the *deferred-shape-array-spec*. The size, bounds and shape of an allocatable array are undefined until the array is allocated by the execution of an ALLOCATE statement (see Section 6.3). Before the array is allocated, no part of the array can be defined, nor can any part of it be referenced except by the ALLOCATED intrinsic function. The lower and upper bounds of each dimension are specified in the ALLOCATE statement when the array is allocated. An allocated array can be deallocated with the DEALLOCATE statement (see Section 6.4).

An example of a specification for a two-dimensional deferred-shape array:

```
REAL,ALLOCATABLE :: D(:,:)
```

5.4.7 POINTER Attribute

An array can be given the POINTER attribute in a type declaration statement, optionally using an array-shape specification with the array name. If the array shape specification is supplied, it must be a *deferred-shape* specification (see Section 5.2.3). This form (R5-50) of the type declaration statement is a special case of the attributed form of type declaration statement (R5-29).

A (POINTER-attributed) *type-decl-stmt* is: R5-50
type-spec, *pointer-attr-spec* :: *entity-decl(deferred-shape-array-spec)*

A *pointer-attr-spec* is: R5-51
POINTER

A *deferred-shape-array-spec* is: R5-8
: [, :]...

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

The rank of the array is equal to the number of colons in the *deferred-shape-array-spec*.

- The `INTENT` attribute must not be specified for an array with the `POINTER` attribute.
- If the `DIMENSION` or `ARRAY` attribute for an array name has been specified previously in the scoping unit, the array-shape specification must have been a *deferred-shape-array-spec*.

5.5 Attribute Specification Statements

Attributes of an entity may be specified by individual attribute specification statements, with the same effect as would be obtained using an attribute specifier in a type declaration statement. An entity must not be given explicitly any attribute more than once in a program unit or interface block.

The attribute specification statements include `DIMENSION` statements, `PARAMETER` statements, `SAVE` statements, `INTENT` statements, `ALLOCATABLE` statements, `POINTER` statements, and `DATA` statements. The first six are described in the following sections; `DATA` statements are described in Section 9.1.

5.5.1 DIMENSION Statement

The `DIMENSION` statement (R5-52) declares the dimensions of one or more array objects.

A *dimension-stmt* is: R5-52
`DIMENSION` array-*name* `(` *array-spec* `)` *[* `,` array-*name* `(` *array-spec* `)` *]*...

Each object named in the list is given the `DIMENSION` attribute and the specified rank and shape.

Examples of `DIMENSION` statements:

```
SUBROUTINE SUB( A, B, C, D )
DIMENSION A(10), B(10,70)  ! explicit-shape real arrays
DIMENSION C(-3:12,*)       ! assumed-size real array
DIMENSION D(:,:,:,:)       ! assumed-shape real array
```


Copyright © 1989–1994 Thinking Machines Corporation

5.5.2 PARAMETER Statement

The PARAMETER statement (R5-53) defines one or more named constants. Named constants defined by a PARAMETER statement have the same properties and restrictions as those declared in a type declaration statement specifying a PARAMETER attribute.

A *parameter-stmt* is: R5-53
PARAMETER (*named-constant-def(s)*)

A *named-constant-def* is: R5-54
name = constant-*expr*

Each *name* is given the PARAMETER attribute and is defined with the value of the associated constant-*expr*, in accordance with the rules of assignment.

The type of a *named-constant* is determined by the implicit typing rules currently in effect for the program unit or interface block, or by the appearance of the name in type specification statements preceding the PARAMETER statement. If the named constant is typed by a subsequent type declaration statement, that type must match the type assumed for the name in the PARAMETER statement.

Examples of PARAMETER statements:

```
PARAMETER ( NUM_STATES = 50, NUM_SENATORS = 2*NUM_STATES )
PARAMETER ( MAJORITY = NUM_SENATORS/2 + 1 )
```

5.5.3 SAVE Statement

The SAVE statement (R5-55) allows variables to be given the SAVE attribute so that they will retain their value from one invocation of a procedure to another. Variables declared local to a subroutine can be saved, as can global variables declared in a common block.

A *save-stmt* is one of: R5-55
- SAVE
- SAVE *[::] saved-object(s)*

A *saved-object* is one of: R5-56
- object-*name*
- / common-block-*name* /

♦ An object-*name* must not be a dummy argument name, a procedure name, a function result name, an adjustable array, an automatic array, or the name of an object in a common block.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

- If a SAVE statement with no saved object list occurs in a program unit, no other SAVE statement (or type declaration statement specifying the SAVE attribute) is permitted in the same program unit.

A SAVE statement with no list is treated as though it contained the names of all allowable items in that program unit, including all accessible common blocks. A SAVE statement appearing in the specification part of a main program or in an interface block has no effect (since all objects declared in the main program exist for the duration of program execution).

All objects named explicitly in a SAVE statement or included within a common block named explicitly in a SAVE statement have the SAVE attribute. The appearance of a common block name preceded and followed by a slash in a SAVE statement has the effect of specifying all entities in that common block.

If a particular common block name is specified in a SAVE statement in any program unit of a program, it must be specified in a SAVE statement in every program unit in which that common block appears (although it need not appear in a SAVE statement in the main program).

For a named common block declared in a SAVE statement, the current values of the objects in the associated common block storage sequence at the time a RETURN or END statement is executed are retained and made available to other procedures that specify the common block name or access the common block either directly or indirectly. If a named common block is specified in the main program, the current values of the common block storage sequence are made available to each program unit that specifies the named common block.

An example of a SAVE statement:

```
SAVE A, B, C, /BLOCKA/, D
SAVE :: X, Y
```

5.5.4 INTENT Statement

An INTENT statement (R5-57) specifies the intended use of the listed dummy arguments of a procedure. (Procedures and dummy arguments are described in Chapter 12.)

An *intent-stmt* is: R5-57

INTENT (*intent-spec*) [::] dummy-arg-*name(s)*

Copyright © 1989–1994 Thinking Machines Corporation

An *intent-spec* is one of: R5-46

- IN
- OUT
- INOUT

- An *intent-stmt* may appear only in a subprogram or an interface block.
- A dummy-arg-*name* must not be the name of a dummy procedure.

Each dummy argument named in the dummy-arg-*name(s)* list is given the INTENT attribute.

The example below contains three INTENT statements:

```
SUBROUTINE DOIT( IN1, IN2, UPD, RES )
INTENT(IN) :: IN1, IN2
INTENT(INOUT) UPD
INTENT(OUT) RES
...
```

The dummy arguments IN1 and IN2 may be read but not modified by statements within the subroutine, and the dummy UPD may be read or changed. The dummy RES may be defined within the subroutine, after which point it may be read.

5.5.5 ALLOCATABLE Statement

The ALLOCATABLE statement (R5-58) specifies a list of *deferred-shape* arrays (see Section 5.2.3) having the ALLOCATABLE attribute. Arrays defined by an ALLOCATABLE statement have the same properties and restrictions as those declared in a type declaration statement specifying an ALLOCATABLE attribute (R5-48).

An *allocatable-stmt* is: R5-58

ALLOCATABLE [::] *array-name* [(*deferred-shape-array-spec*)]
[, *array-name* [(*deferred-shape-array-spec*)]]...

A *deferred-shape-array-spec* is: R5-8

: [, :]...

The type of the array is determined by the implicit typing rules currently in effect for the program unit or interface block, or by the appearance of the array name in type specification statements preceding the ALLOCATABLE statement.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

An example of an ALLOCATABLE statement:

```
REAL A, B(:)
ALLOCATABLE :: A(:,:), B
```

5.5.6 POINTER Statement

The POINTER statement (R5-59) specifies a list of objects (array names) that have the POINTER attribute, optionally adding an array-shape specification. If the array-shape specification is supplied, it must be a *deferred-shape* specification.

A *pointer-stmt* is: R5-59

POINTER [::] *array-name* [(*deferred-shape-array-spec*)]
[, *array-name* [(*deferred-shape-array-spec*)]]...

A *deferred-shape-array-spec* is: R5-8

: [, :]...

The rank of the array is equal to the number of colons in the *deferred-shape-array-spec*.

- The INTENT attribute must not be specified for *array-name*.
- If the DIMENSION or ARRAY attribute for an *array-name* has been specified previously in the scoping unit, the array-shape specification must have been a *deferred-shape-array-spec*.

5.6 IMPLICIT Statement

An IMPLICIT statement (R5-60) associates a type with a letter or a range of letters to establish implicit typing rules in effect for a region of a program unit or interface block. The form IMPLICIT NONE indicates that no implicit typing rules are to apply within the containing program unit or interface block.

An *implicit-stmt* is one of: R5-60

- IMPLICIT *implicit-spec(s)*
- IMPLICIT NONE

An *implicit-spec* is: R5-61

type-spec (*letter [– letter]*)

- If the second *letter* is present, it must follow the first *letter* alphabetically.

Copyright © 1989–1994 Thinking Machines Corporation

- If IMPLICIT NONE is specified in a scoping unit (a program unit or interface block), it must precede any PARAMETER statements that appear in the scoping unit, and there must be no other IMPLICIT statements in the scoping unit (except possibly within nested interface blocks).

Each *implicit-spec* (R5-61) in an IMPLICIT statement associates one or more letters of the alphabet with the data type given in the type specification. Two letters separated by a minus in an *implicit-spec* specifies a range of letters, and has the same effect as individually associating each of the letters in the alphabetic sequence from the first letter through the second letter. For example,

```
IMPLICIT REAL (K-M)
```

is the same as

```
IMPLICIT REAL (K), REAL (L), REAL (M), REAL (N)
```

The same letter must not be associated with a type more than once in all the IMPLICIT statements in a given program unit or interface block.

An implicitly-typed entity is any entity other than an intrinsic function that is not explicitly declared by a type declaration statement. In a program unit or interface block containing an IMPLICIT statement with an *implicit-spec*, all implicitly-typed entities whose names begin with the letters specified in the IMPLICIT statement and which are introduced *after* the appearance of the IMPLICIT statement are considered to have the type associated with the first letter of the name.

In a program unit, if an association is not specified for a letter, the default is the association of letters with type specifications given by the declaration

```
IMPLICIT INTEGER (I-N), REAL (A-H, O-Z)
```

In an interface block, if an association is not specified for a letter, the default is the association in the containing program unit or interface block.

IMPLICIT NONE specifies that there is no association for any letters, effectively requiring all entities in a program unit or interface block to be declared. The `-implicit_none` switch of the `cmf` command can be used to change the default association to IMPLICIT NONE. See the *CM Fortran User's Guide* for information regarding the `cmf` command switches.

Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation

Some examples of the use of IMPLICIT statements:

```
SUBROUTINE SCOPE1
PARAMETER ( K = 1 )            ! K is integer!!
IMPLICIT COMPLEX (K), REAL (I)
PARAMETER ( KK = K )           ! KK is complex!!
PARAMETER ( I = 1.7, J = 4 )   ! I is real; J is integer
...
END SUBROUTINE SCOPE1

SUBROUTINE SCOPE2
IMPLICIT NONE
INTERFACE
  FUNCTION FUN(I)
  INTEGER FUN, I           ! All entities must be declared
END INTERFACE
INTERFACE
  IMPLICIT COMPLEX (C)
  SUBROUTINE SUB(CC,R)
  DIMENSION CC(100)        ! CC is implicitly complex
  REAL R                   ! R is real (and must be declared)
END INTERFACE SUB
...
REAL, DIMENSION(10) :: I  ! All entities must be declared
INTEGER J
...
END SUBROUTINE SCOPE2
```

5.7 NAMELIST Statement

The NAMELIST statement associates a set of variables with a namelist name. This name can be used in READ or WRITE statements to transfer data to or from the set of variables. See Section 13.4.1.3 for details of this use.

A *namelist-stmt* is: R5-62

NAMELIST */namelist-name/*
namelist-object(s)
[[,] /namelist-name/
namelist-object(s)]...

A *namelist-object* is: R5-63

variable

Copyright © 1989–1994 Thinking Machines Corporation

- A *namelist-object* must not be an array element, array section, or substring. It must not be an allocatable array, a pointer, a dummy argument with non-constant bounds, a pointer, a variable with assumed character length, or an automatic object.

Any *namelist-name* may occur in more than one NAMELIST statement in a program unit. The list of *namelist-objects* following each successive appearance of the same *namelist-name* in a program unit is treated as a continuation of the list for that *namelist-name*.

The order in which the variables are specified in the NAMELIST statement determines the order in which the values appear on output.

A namelist object either must have its type and shape specified by a previous occurrence in a type declaration statement in the same program unit, or must be determined by the implicit type rules currently in effect for the program unit. If a namelist object is typed by the implicit typing rules, its appearance in any subsequent type declaration statement must confirm this implied type.

An example of a NAMELIST statement is

```
NAMELIST /NLIST/ A, B, C
```


Version 2.2, October 1994
Copyright © 1989–1994 Thinking Machines Corporation