## Chapter 7

# Expressions

An *expression* (R7-1) is a data reference or a computation whose value is a scalar or an array. An expression may be *evaluated* during program execution, and the result of the evaluation has type and shape as well as value. An expression is formed from operators, primaries, and parentheses.

An *expr* is one of: R7-1
- *primary*
- *operation*

An expression that involves no operators is called a *primary*. An expression that involves one or more binary or unary operators and one or more *operands* that are primaries is an *operation*.

An expression is a *numeric expression*, *character expression*, or *logical expression*, depending on whether its type is numeric, character, or logical, respectively. An expression is a *relational expression* if it is an operation involving one or more of the *relational operators*. All these forms of expressions are defined in the following sections.

## 7.1 Primaries

The most fundamental form of expression is the *primary* (R7-2).

A *primary* expression is one of: R7-2
- *constant*
- *variable*
- *array-constructor*
- *function-reference*
- ( *expr* )


*Copyright © 1989–1994 Thinking Machines Corporation*

A primary is a *numeric primary*, *character primary*, or *logical primary*, depending on whether its type is numeric, character, or logical, respectively. These forms of primary are defined in the following sections. The type and shape of a primary are determined by its kind.

Any variable or function referenced in an expression must be defined at the time the reference is executed. An integer variable used in an expression must be defined with an integer value rather than a statement label value.

Examples of a *primary*:

```
1.                ! literal constant
A                 ! variable (or named constant)
[1.,2.,3.]        ! array constructor
F( X, Y )         ! function reference
(X+3)             ! parenthesized expression
```

### 7.1.1 Constants

A *constant* (R7-3) is an arithmetic constant, logical constant, or character constant. The value of a constant does not change. Within a program, all constants that have the same form have the same value.

A *constant* is one of: R7-3
- *literal-constant*
- *named-constant*

A *named-constant* is: R7-4
scalar-constant-*name*

A *literal constant* (R4-1, p. 24) is a scalar whose value and type can be determined directly from its syntactic form. A *named constant* (R7-4) is a name that has been associated with a literal constant using the `PARAMETER` statement (or using the `PARAMETER` attribute in a type declaration statement). A named constant is denoted by its name. A reference to a constant is always permitted; redefinition of the value of a constant is never permitted.

### 7.1.2 Variables

A *variable* (R7-5) is an object whose value may be changed during program execution. Any object not having the `PARAMETER` attribute is a variable, regardless of whether it is declared explicitly or implicitly.


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

A *variable* is one of: R7-5
- variable-*name*
- *variable-array-element-designator*
- *variable-array-section*
- *variable-substring-designator*

A *variable-array-element-designator* is: R7-6
array-variable-*name* ( *subscript(s)* )

A *variable-array-section* is: R7-7
array-variable-*name* ( *section-subscript(s)* )

A *variable-substring-designator* is one of: R7-8
- scalar-variable-*name* ( *substring-range* )
- array-variable-*name* ( *subscript(s)* ) ( *substring-range* )

A variable may be a scalar, a whole array, an array element, an array section, or a substring of a string. The data type of a variable determines which values that variable may take. Assignment provides one means of defining or redefining the value of a variable of any type. The data type of a variable determines the operations that may be used to manipulate the variable.

### 7.1.3 Substrings

A *substring* is a contiguous subsequence of characters obtained from a character string using a *substring designator* (R7-9). A substring is a variable if its *parent-string* (R7-10) is a variable, and is a constant if its *parent-string* is a constant.

A *substring-designator* is: R7-9
*parent-string* ( *substring-range* )

A *parent-string* is one of: R7-10
- scalar-variable-*name*
- scalar-constant-*name*
- *array-element-designator*

A *substring-range* is: R7-11
*[ starting-point ]* **:** *[ ending-point ]*

A *starting-point* is: R7-12
scalar-integer-*expr*

An *ending-point* is: R7-13
scalar-integer-*expr*


Copyright © 1989–1994 Thinking Machines Corporation

- The *parent-string* must be of type character.

The *starting-point* and *ending-point* of a substring range indicate the first and last characters of the parent string to be included in the substring. The *length* of a substring is the number of characters in the substring, and is equal to *ending-point* − *starting-point* + `1`.

If the characters in the parent string are numbered `1,2,3,...,`*n*, where *n* is the length of the parent string, then the characters in the substring are those from the parent string beginning at the starting point and proceeding in sequence up to and including the ending point. Both the starting point and the ending point must be within the range `1,2,...,`*n*, and the *starting-point* must be less than or equal to the *ending-point*. If the starting point is not specified, the default value is `1`. If the ending point is not specified, the default value is *n*.

Evaluation of a substring designator entails the evaluation of its substring range expressions, if any. The type of an expression in which a substring designator appears does not affect and is not affected by the evaluation of its range expressions.

Examples of character substrings:

```
ID(4:9)     ! characters 4 - 9 of scalar character variable ID
B(2)(1:5)   ! characters 1 - 5 of element 2 of array B
```

### 7.1.4 Array Elements

An array element is a scalar, and is specified using an *array element designator* (R7-14) naming the array and the element referenced. An array element is a variable.

An *array-element-designator* is: R7-14
parent-array-*name* ( *subscript(s)* )

A *subscript* is: R7-15
scalar-integer-*expr*

- The number of subscripts in the *subscript(s)* list of an *array-element-designator* must equal the declared rank of the parent array.

The value of a *subscript* (R7-15) must be within the declared bounds for the dimension being subscripted.

Execution of an array element designator requires the evaluation of all its subscripts. When a reference to an array element is made, the selected element must be defined. The type of an expression in which the array element designator appears does not affect and is not affected by the evaluation of its subscripts.


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

Given the declarations

```
REAL A(10,10), B(30,20)
```

`A(1,2)` and `B(29,19)` are array elements.

### 7.1.5 Array Element Ordering

The elements of an array can be regarded as being ordered in a sequence known as the *array element ordering*. For an array `A` with shape `[m,n]`, pictured in two dimensions as:

```
A(1,1),  A(1,2),..., A(1,n)
A(2,1),  A(2,2),..., A(2,n)
...
A(m,1),  A(m,2),..., A(m,n)
```

the elements in array element order are:

```
A(1,1)
A(2,1)
...
A(m,1)
      A(1,2)
      A(2,2)
      ...
      A(m,2)
              ...
                   A(1,n)
                   A(2,n)
                   ...
                   A(m,n)
```

This ordering is sometimes called *column-major ordering*.

The position of an array element in this linear sequence is determined by the *subscript order value* of the *subscript(s)* list designating the element. The subscript order value is computed from the formulas in Table 1, page 76.


Copyright © 1989–1994 Thinking Machines Corporation

**Table 1. Subscript order value.**

| Rank | Explicit-Shape Specifier | Subscript List | Subscript Order Value |
|---|---|---|---|
| 1 | $j_1 : k_1$ | $s_1$ | $1 + ( s_1 - j_1 )$ |
| 2 | $j_1 : k_1 , \; j_2 : k_2$ | $s_1 , \; s_2$ | $1 + ( s_1 - j_1 ) + ( s_2 - j_2 ) * d_1$ |
| 3 | $j_1 : k_1 , \; j_2 : k_2 , \; j_3 : k_3$ | $s_1 , \; s_2 , \; s_3$ | $1 + ( s_1 - j_1 ) + ( s_2 - j_2 ) * d_1 + ( s_3 - j_3 ) * d_2 * d_1$ |
| ... | ... | ... | ... |
| 31 | $j_1 : k_1 , j_2 : k_2 , \ldots , j_{31} : k_{31}$ | $s_1 , s_2 , \ldots , s_{31}$ | $1 + ( s_1 - j_1 ) + ( s_2 - j_2 ) * d_1 + ( s_3 - j_3 ) * d_2 * d_1 \; \ldots \; + ( s_{31} - j_{31} ) * d_{30} * d_{29} * \ldots * d_1$ |

Notes:
- $d_i$ = MAX ( $k_i - j_i + 1 , 0$ ) is the size of the *i*th dimension.
- If the size of the array is nonzero, $j_i \le s_i \le k_i$ for all $i = 1, 2, \ldots, 31$.

### 7.1.6 Array Sections

An *array section* (R7-16) is a subset of the elements of an array, designated by an array name followed by a list of one or more *section subscripts* (R7-17) in parentheses. An array section is a variable.

An *array-section* is: R7-16
parent-array-*name* ( *section-subscript(s)* )

A *section-subscript* is one of: R7-17
- *subscript*
- *subscript-triplet*
- *vector-integer-expr*

A *subscript-triplet* is: R7-18
*[ subscript ]* : *[ subscript ] [ : stride ]*

A *subscript* is: R7-15
scalar-integer-*expr*


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

A *stride* is: R7-19
scalar-integer-*expr*

A *vector-integer-expr* is: R7-20
*expr*

- At least one *section-subscript* of a *section-subscript(s)* list must be a *subscript-triplet* or a *vector-integer-expr* (otherwise, the construct is a scalar *array-element-designator* (R7-14) rather than an array section).
- A *vector-integer-expr* of a *section-subscript* must be a rank-one integer array expression.
- The number of *section-subscript*s in an array section must equal the declared rank of the parent array.

Each subscript triplet (R7-18) and each vector integer expression (R7-20) in a *section-subscript(s)* list represents a *sequence* of subscripts in the array dimension in which it appears. The set of elements represented by an array section is determined by all possible combinations of array elements obtainable from the single subscripts or sequences of subscripts specified by each section subscript.

The *rank* of an array section is the total number of section subscripts in the section that are either subscript triplets or vector expressions. The *shape* of the section is the rank-one array whose *i*th element is the number of integer values in the sequence indicated by the subscript triplet or the vector expression. If any of these sequences is empty, the array section has size zero. The subscript order of the elements of an array section is that of the array object that the array section represents.

The value of each subscript in a section subscript must lie within the declared bounds for the dimension in which it appears, unless a subscript triplet or a vector integer expression in the same section subscript defines an empty sequence.

Use of an array section requires the evaluation of its section subscripts. The type of expression in which an array section appears does not affect and is not affected by the evaluation of the array section subscripts. When an array section is evaluated, all the selected elements must be defined. Note that it is not necessary for a program to evaluate any subscript expressions for a parent array of zero size.

#### 7.1.6.1 Subscript Triplets

The subscripts and stride of a subscript triplet (R7-18) are optional (but the colon must be present). A subscript triplet with an omitted first subscript is equivalent to a subscript triplet whose first subscript value is the declared lower bound for the array; an omitted second


Copyright © 1989–1994 Thinking Machines Corporation

subscript is equivalent to a subscript whose value is the declared upper bound. An omitted stride is equivalent to a stride of one. In an array section specifying an assumed-size parent array, the second subscript must not be omitted in the subscript triplet for the last dimension.

A subscript triplet designates a regular sequence of subscripts consisting of zero or more subscript values. The first expression in a subscript triplet is the *lower bound*, the second is the *upper bound*, and the third expression is the increment between successive subscript values, and is called the *stride*. The *subscript* and *stride* expressions of a subscript triplet are evaluated to form a sequence of integers:

- When the stride is positive, the subscripts specified by a triplet form a regularly spaced sequence of integers beginning with the first subscript and proceeding in successive increments of the stride to the largest such integer not exceeding the second subscript; the sequence is empty if the first subscript exceeds the second.
- The stride must not be zero.
- When the stride is negative, the sequence begins with the first subscript and proceeds in increments of the stride down to the smallest such integer equal to or exceeding the second subscript; the sequence is empty if the second subscript exceeds the first.

For example, if an array `B` is declared as `B(12)`, the array section `B(3:11:7)` is the array of shape `[2]` consisting of the elements `B(3)` and `B(10)`, in that order. The section `B(9:1:-2)` is the array of shape `[5]` whose elements are `B(9)`, `B(7)`, `B(5)`, `B(3)`, and `B(1)`, in that order.

For another example, if an array `A` is declared as `A(5,4,3)`, then the section `A(3:5,2,1:2)` is the array of shape `[3,2]` shown below:

```
A(3,2,1)    A(3,2,2)
A(4,2,1)    A(4,2,2)
A(5,2,1)    A(5,2,2)
```


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

**NOTE**

The following example illustrates an important distinction between a subscript and a subscript triplet in an array section. Given the declarations

```
REAL A(10,10)
INTEGER B(5,5,5)
```

then

- `A(1:N,J)` is a vector with shape `[N]`
- `A(1:N:2,M)` is a vector of length `N/2`
- `A(1:N,J:J)` is an array with shape `[N,1]`
- `B(:,:,3)` is an array with shape `[5,5]`
- `B(1:1,1:1,1:1)` is an array with shape `[1,1,1]`
- `B(1,1:1,1)` is an array with shape `[1]`

### 7.1.6.2 Vector-Valued Subscripts

Vector-valued subscripts provide a means to choose a subset of elements from an array in an arbitrary order. The use of a vector integer expression as a section subscript (R7-17) in an array section (R7-16) designates a *sequence* of subscripts that are the values of the expression; each element of the expression must be defined. The sequence is empty if the expression is of size zero.

Vector-valued subscripts in array sections appearing on the left-hand side of an assignment statement must not index a particular array element more than once. This is to prevent "collisions" that would cause the duplicated element to be assigned a value more than once. (The values of any duplicated elements are indeterminate since array assignment does not specify the order in which array elements are assigned. The error is not detected either at compile time or at run time.)


Copyright © 1989–1994 Thinking Machines Corporation

As an example of vector-valued subscripting, suppose Z is a two-dimensional array of shape [5,7] and U and V are one-dimensional arrays of shape [3] and [4] respectively. Assume the values of U and V are

```
U = [1,3,2]
V = [2,1,1,3]
```

Then Z(3,V) consists of the elements from the third row of Z in the order:

```
Z(3,2) Z(3,1) Z(3,1) Z(3,3)
```

and Z(U,2) consists of the column elements:

```
Z(1,2) Z(3,2) Z(2,2)
```

and Z(U,V) consists of the elements:

```
Z(1,2) Z(1,1) Z(1,1) Z(1,3)
Z(3,2) Z(3,1) Z(3,1) Z(3,3)
Z(2,2) Z(2,1) Z(2,1) Z(2,3)
```

Because Z(3,V) and Z(U,V) contain duplicate elements from Z, neither may be used as a section on the left side of an assignment statement.

**NOTE**

The following example illustrates an important distinction between a subscript and a vector-valued subscript in an array section. Given the program fragment

```
REAL A(10,10)
INTEGER B(5,5,5), V(4), W(1)
...
V = [3,5]
W = [2]
```

then

- A(V,J) is a vector with shape [4]
- A(V,W) is an array with shape [4,1]
- B(V,V,3) is an array with shape [4,4]
- B(W,V,W) is an array with shape [1,4,1]
- B(1,V,1) is an array with shape [4]



Copyright © 1989–1994 Thinking Machines Corporation

### 7.1.7 Array Constructors

The *array constructor* (R7-21) provides a means of specifying a rank-one array-valued expression as an arbitrary sequence of scalar values. The sequence of values may be specified by any combination of individual scalar expressions, ranges of values, rank-one arrays, and other array constructors. When used in an assignment statement, each scalar value in the sequence must be convertible to the type of variable to which the array is being assigned, and is converted using the rules of assignment conversion (Section 10.1.1). Array constructors are passed in procedure references as CM arrays.

An *array-constructor* is one of: R7-21
- `[` *array-constructor-value(s)* `]`
- `(/` *array-constructor-value(s)* `/)`

An *array-constructor-value* is one of: R7-22
- scalar-*expr*
- *vector-integer-expr*
- *subscript* `:` *subscript* `[` `:` *stride* `]`
- scalar-integer-*expr array-constructor*

A *subscript* is: R7-15
scalar-integer-*expr*

A *stride* is: R7-19
scalar-integer-*expr*

A *vector-integer-expr* is: R7-20
*expr*

- ♦ A *vector-integer-expr* forming an *array-constructor-value* must be a rank-one integer array expression.
- ♦ An *array-constructor-value* must not be a character expression.

The type of an array constructor is the type of its first *array-constructor-value*.

The four forms of *array-constructor-value* (R7-22) are described below.

- ■ The first form is a scalar expression that represents the value of a single element of the constructed array with the specified value.

  Examples of array constructors with constructor values of this form:

```
(/ 0, 1, 2, 4, 8, 16, 32, 64, 128 /)
[1,(2,0),(0.,1.)]    ! an integer vector of length 3
[1,7,-2,333,-48]     ! an integer vector of length 5
```


Copyright © 1989–1994 Thinking Machines Corporation

```
[(0.,1.),1,(2,0)]    ! a complex vector of length 3
```

- The second form is a rank-one expression interpreted as a sequence of scalar values comprising values of successive array elements in array element order. (This expression can itself be an *array-constructor*.)

  Examples of array constructors with constructor values of this form are (assuming that ivector is a rank-one integer expression):

```
[ ivector, [ 256, 512, 1024 ] ]

(/ [ 0, 1, 2, 4, 8 ], [ 16, 32, 64, 128 ] /)

[A]   ! same as A, assuming that A is a vector
```

  The second example has the same value as the constructor given in the previous example.

- The third form, with a syntax like that of a subscript triplet in an array section (R7-16), specifies a sequence of values as a rank-one array. The *stride*, if present, must not be zero; if absent, the default *stride* is 1.

  When the *stride* is positive, the values specified form a regularly spaced sequence of integers beginning with the first *subscript* and proceeding in increments of the *stride* to the largest such integer not exceeding the second *subscript*; the sequence is empty if the first *subscript* exceeds the second.

  When the stride is negative, the sequence of integers begins with the first *subscript* and proceeds in increments of the *stride* down to the smallest such integer equal to or exceeding the second *subscript*; the sequence is empty if the second *subscript* exceeds the first.

  An empty sequence formed by a subscript triple yields a zero-sized, rank-one array.

  Examples of array constructors with constructor values of this form:

```
[1:100]        ! vector of integers: 1 to 100

(/ 1:100:2 /)   ! vector of odd integers: 1 to 99

[ [1:100:2], [101:200:2] ] ! odd integers: 1 to 199
```

- The scalar-integer-*expr* in the fourth form of *array-constructor-value* specifies the number of consecutive copies of the following *array-constructor*.

  Examples of array constructors with constructor values of this form:

```
[ 100 [5*10] ]      ! vector of 100 fifties
```



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

```
[ 10 [1:100] ]      ! vector of 1000 integers
                    !   1,2,...,99,100,1,2...,99,100...
```

A rank-one array may be constructed from scalars and other rank-one arrays and may be reshaped into any allowable array shape using the RESHAPE intrinsic function (Chapter 17). For example, the assignments

```
X = (/ 3.2, 4.01, 6.5 /)
Y = RESHAPE( MOLD = [3,2], SOURCE = [2.,2[4.5],X] )
```

result in Y having the 3 by 2 array of values:

```
2.0  3.2
4.5  4.01
4.5  6.5
```

**NOTE:** When a program is compiled with the `-f90syntax` switch, the keyword SHAPE is used instead of the equivalent MOLD, and the argument order is changed, in order to make the function RESHAPE compatible with Fortran 90 (See Chapter 17.)

## 7.2 Operations and Operators

An expression involving one or more operators is called an *operation* (R7-23). The operators of an expression may be any of the *unary operators* (R7-24) or the *binary operators* (R7-25) defined below. A *unary operator* accepts a single operand, and a *binary operator* accepts a pair of operands. A *unary operation* is an expression of the form *u-op* $x_2$, where *u-op* is a unary operator and $x_2$ is an operand. A *binary operation* is an expression of the form $x_1$ *b-op* $x_2$, where $x_1$ and $x_2$ are operands. The type of an operation depends on the operator and on the types of the operands used in the operation. Table 2, page 86, gives the rules for determining the result type of an operation given the types of its operands.

An operand is a primary (R7-2) that is used in an operation. An operand is either a scalar or an array; array operands of a binary operation must be conformable (see Section 7.2.1). An expression may combine several operations, and more complicated expressions are formed using operands that are themselves expressions.

An *operation* is one of: R7-23

- *u-op primary [ b-op primary ]...*
- *primary b-op primary [ b-op primary ]...*


Copyright © 1989–1994 Thinking Machines Corporation

A *u-op* is one of: R7-24

- `+` ! identity
- `-` ! negation
- `.NOT.` ! logical negation; bitwise complementation for integers

A *b-op* is one of: R7-25

- `**` ! exponentiation
- `*` ! multiplication
- `/` ! division
- `+` ! addition
- `-` ! subtraction
- `//` ! concatenation
- `.EQ.` • `==` ! equal to
- `.NE.` • `/=` ! not equal to
- `.LT.` • `<` ! less than
- `.LE.` • `<=` ! less than or equal to
- `.GT.` • `>` ! greater than
- `.GE.` • `>=` ! greater than or equal to
- `.AND.` ! logical conjunction; bitwise conjunction
- `.OR.` ! logical inclusive disjunction; bitwise inclusive disjunction
- `.XOR.` ! logical exclusive disjunction; bitwise exclusive disjunction
- `.EQV.` ! logical equivalence; bitwise equivalence
- `.NEQV.` ! logical nonequivalence; bitwise nonequivalence

**NOTE:** CM Fortran also accepts the operator <> meaning "not equal to."

**NOTE:** `.XOR.` and `.NEQV.` are identical.

The shape of an expression, like its type, depends on the operators and on the shapes of the primaries used in the expression, and is determined from the syntactic form of the expression. For a unary operator *op*, the shape of a unary operation *op* $x_2$ is the shape of $x_2$; for a binary operator *op*, the shape of the operation $x_1$ *op* $x_2$ is the shape of $x_1$, or the shape of $x_2$ if $x_1$ is a scalar.

Assume the following declarations:

```
REAL, DIMENSION(10,10) :: A, B, C, D ! four arrays, same shape
CHARACTER*5 :: E, G, H*12            ! three strings
LOGICAL :: X, Y, MASK1(128), MASK2(128)
```


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

The following are valid examples of operations involving scalars:

```
E // G .EQ. H(1:10)
.NOT. X .AND. Y
X .AND. MASK1  ! equal to MASK1 if X is true;
               ! equal to [ 128 [.false.] ] otherwise
```

The following are valid examples of operations involving arrays:

```
A + B
(A - B) * C
A ** B
A ** 2             ! square all elements of A
-A + B == C * D    ! does this relation hold?
MASK1 .EQV. MASK2  ! are masks identical?
```


Copyright © 1989–1994 Thinking Machines Corporation

**Table 2. Types of operands and result for the operation [ $x_1$ ] *op* $x_2$.**

The symbols I, K, R, D, Z, Q, S, and L stand for the types integer, INTEGER*8, single-precision real, double-precision real, single-precision complex, double-precision complex, character, and logical, respectively. When 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.

| Operator Class | Operator *op* | Type of $x_1$ | Type of $x_2$ | Type of [ $x_1$ ] *op* $x_2$ |
|---|---|---|---|---|
| Numeric | unary +, − | | I,K,R,D,Z,Q | I,K,R,D,Z,Q |
| | binary +, −, *, /, ** | I | I,K,R,D,Z,Q | I,K,R,D,Z,Q |
| | | K | I,K,R,D,Z,Q | K,K,R,D,Z,Q |
| | | R | I,K,R,D,Z,Q | R,R,R,D,Z,Q |
| | | D | I,K,R,D,Z,Q | D,D,D,D,Z,Q |
| | | Z | I,K,R,D,Z,Q | Z,Z,Z,Z,Z,Q |
| | | Q | I,K,R,D,Z,Q | Q,Q,Q,Q,Q,Q |
| Character | // | S | S | S |
| Relational | .EQ., .NE.<br>==, /= | I | I,K,R,D,Z,Q | L,L,L,L,L,L |
| | | K | I,K,R,D,Z,Q | L,L,L,L,L,L |
| | | R | I,K,R,D,Z,Q | L,L,L,L,L,L |
| | | D | I,K,R,D,Z,Q | L,L,L,L,L,L |
| | | Z | I,K,R,D,Z,Q | L,L,L,L,L,L |
| | | Q | I,K,R,D,Z,Q | L,L,L,L,L,L |
| | | S | S | L |
| | .LT., .LE., .GT., .GE.<br><, <=, >, >= | I | I,K,R,D | L,L,L |
| | | K | I,K,R,D | L,L,L |
| | | R | I,K,R,D | L,L,L |
| | | D | I,K,R,D | L,L,L |
| | | S | S | L |
| Logical | .NOT. | | I,K,L | I,K,L |
| | .AND., .OR., .XOR.<br>.EQV., .NEQV. | I | I,K | I,K |
| | | K | I,K | K,K |
| | | L | L | L |




Copyright © 1989–1994 Thinking Machines Corporation

### 7.2.1 Operations Involving Array Operands

Two operands are said to be *conformable* if both are arrays of the same shape, or both are scalars, or one is an array and the other is a scalar. Two operands $x_1$ and $x_2$ of a binary operation must be conformable. If one operand 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 equal to the value of the scalar.

When a binary operator is applied to a pair of operands and at least one of the operands is an array operand, the operation is performed elementally on corresponding elements of the operands. 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. The element-by-element operations are performed in an arbitrary order (possibly simultaneously).

When a unary operator is applied to an array operand, the operation is performed elementally, in an arbitrary order. The result of the operation is the same shape as the operand.

These remarks regarding elemental operations also apply to the elemental intrinsic functions and array assignment. For example, if `A`, `B`, `C`, `D`, and `X` are conformable arrays, then a call of the form

```
X = MAX( A, B, C, D )
```

to the intrinsic function `MAX` is performed element-by-element on each of the four arrays, and the assignment to `X` of the result is likewise performed element-by-element.

The expression

```
Q + 2.3 * R
```

is a scalar expression if `Q` and `R` are scalars, and an array expression if either `Q` or `R` is an array. If `A` and `B` are arrays, then the following expression is an array expression:

```
A(1:10) + B(2:11)
```

## 7.3 Precedence and Associativity of Operators

There is a *precedence* among the operations of an expression that determines the order in which the operations are performed, unless the order is changed by the use of parentheses.


Copyright © 1989–1994 Thinking Machines Corporation

The precedence of an operation is that of its operator, and the precedence of the operators is summarized in Table 3.

**Table 3. Precedence and associativity of operators.**

| Category | Operator | Precedence | Associativity |
|---|---|---|---|
| Numeric | `**` | Highest | R–L |
| Numeric | `*` or `/` | . | L–R |
| Numeric | + or − (unary) | . | – |
| Numeric | + or − (binary) | . | L–R |
| Character | `//` | . | L–R |
| Relational | `.EQ., .NE., .LT., .LE., .GT., .GE.`<br>`==, /=, <, <=, >, >=` | . | L–R |
| Logical | `.NOT.` | . | – |
| Logical | `.AND.` | . | L–R |
| Logical | `.OR.` | . | L–R |
| Logical | `.XOR.,`<br>`.EQV., .NEQV.` | Lowest | L–R |

Higher precedence operations of an expression are always evaluated before lower precedence operations. For example, in the expression

```
- A + B * C
```

both the (unary) negation operator (−) and the multiplication operator (*) have precedence over the addition operator (+). The interpretation of the expression above is therefore

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

When several binary operations of equal precedence appear in an expression, such as

```
A * B / C * D ** E ** F
```

there are several possible ways in which their operands can be combined. All binary operators except for exponentiation combine their operands from left to right, while exponentiation combines its operands from right to left. This property of an operator is




Copyright © 1989–1994 Thinking Machines Corporation

called its *associativity*, and is listed in Table 3 for all operators. As the table indicates, the expression above has interpretation

```
( ( A * B ) / C ) * ( D ** ( E ** F ) )
```

because of the relative precedence and associativity of the operators involved.

Note that the syntax rules for expressions do not permit expressions containing two consecutive numeric operators, such as `A**-B`, although the form `A**(-B)` is permitted.

## 7.4 Evaluation and Interpretation of Operands

The CM Fortran compiler may not cause all operands of an expression to be evaluated if the value of the expression can be determined otherwise. In practice, this rule most often applies to logical expressions and zero-sized arrays, but can apply to any expression. For example, in evaluating the expression

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

where `L` is a logical function, and `X`, `Y`, and `Z` are real, the subexpression `X .GT. Y` need not be evaluated if the function reference to `L` returns true. Similarly, in the array expression

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

where `X` is a zero-size array and `W` is a function, the function reference `W(Z)` may not be evaluated. If a statement contains a function reference in 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. In the preceding examples, evaluation of these expressions causes `Z` to become undefined if `L` or `W` defines its argument.

The type and interpretation of an expression that consists of the application of an operator to a single operand or a pair of operands are independent of the context in which the expression appears. In particular, the type and interpretation of such an expression are independent of the type of any larger expression in which it occurs. For example, if `X` is of type real, `J` is of type integer, and `INT` is the intrinsic real-to-integer conversion function, the expression `INT( X + J )` is an integer expression and `X + J` is a real expression.


Copyright © 1989–1994 Thinking Machines Corporation

## 7.5 Evaluation of Parenthesized Expressions

In the absence of parentheses, the compiler may take advantage of the distributive and associative properties of mathematically equivalent operations in order to optimize generated code. Two expressions of a given numeric type are *mathematically equivalent* if their values are equal for all possible values of their primaries. Mathematically equivalent expressions of numeric type may produce different computational results. For example, any difference between the values of the expressions `(1./3.)*3.` and `1.` is a computational difference, not a mathematical difference.

An expression contained in parentheses is treated as a single unit by the compiler, and its value may be used as if it were a single data entity within any larger expression in which it appears. Parentheses effectively restrict the possible order in which operands of an expression may be combined when evaluating an expression. This is useful for controlling the magnitude and accuracy of intermediate values obtained during the evaluation of an expression. For example, in evaluating the expression `A + (B – C)` where `A`, `B`, and `C` are of numeric types, the difference of `B` and `C` must be evaluated before the addition operation is performed; the compiler will not evaluate the mathematically equivalent expression `(A + B) – C`.

## 7.6 Numeric Expressions

A *numeric expression* (R7-26) is an expression (R7-1) that expresses a numeric computation. Evaluation of a numeric expression produces a numeric value.

A *num-expr* is one of: R7-26
- *num-primary*
- *num-operation*

A *num-operation* is one of: R7-27
- *num-u-op num-primary [ num-b-op num-primary ]...*
- *num-primary num-b-op num-primary [ num-b-op num-primary ]...*

A *num-primary* is one of: R7-28
- num-*constant*
- num-*variable*
- num-*array-constructor*
- num-*function-reference*
- ( *num-expr* )


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

A *num-u-op* is one of: R7-29

- + ! identity
- − ! negation

A *num-b-op* is one of: R7-30

- ** ! exponentiation
- * ! multiplication
- / ! division
- + ! addition
- − ! subtraction

A *numeric operation* (R7-27) is a numeric expression in which the operator is one of the *numeric operators* defined above and whose operands are *numeric primaries* (R7-28). For numeric binary operations, the two operands may be of different numeric types. Except for a value raised to an integer power, if the operands do not have the same types, each operand that differs in type from that of the result is converted to the type 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 is not converted.

**Table 4. Interpretation of the numeric operators.**

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

The permitted data types for operands of the numeric operators are specified in Table 2, page 86. The numeric operators and their interpretation in an expression are given in Table 4 for expressions of the form $x_1$ *op* $x_2$ for binary operations and of the form *op* $x_2$ for unary operations.

The exponentiation operation $x_1$ ** $x_2$ for integer $x_1$ and $x_2$ and negative $x_2$ has the same interpretation as `1 / (` $x_1$ `** ABS(` $x_2$ `))`, and is subject to the rules of integer division. For example, `2**(-3)` has the value `1/(2**3)`, which is zero. The operation $x_1$ ** $x_2$ for complex $x_1$ and $x_2$ has the same interpretation as the expression


Copyright © 1989–1994 Thinking Machines Corporation

EXP( $x_2$*LOG($x_1$)), where EXP and LOG are intrinsic functions described in Chapter 17. The value of the operation is the "principal value" determined by the reference to EXP.

The interpretation of a division operation depends on the data types of its operands. If both operands are of type integer, the result is of type integer even though the mathematical quotient is not necessarily an integer. The result is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively. For example, the expression (-8)/3 has the value (-2).

Any numeric operation whose result is not mathematically defined is prohibited in the execution of a program. Examples of such an operation are division by zero and raising a zero-valued primary to a zero-valued or negative-valued power. Raising a negative-valued primary of type real to a real power is also prohibited and will result in a run-time error.

Simple examples of numeric expressions:

```
3.14159
A
A(I,J)
B**C
D * E
+1
F - 1.
```

## 7.7 Character Expressions

A *character expression* (R7-31) specifies a character value or the concatenation of two or more character values. The character operator // concatenates two operands of type character. Evaluation of the concatenation operation produces a result of type character. A character expression is a form of expression (R7-1).

A *char-expr* is one of: R7-31
- *char-primary*
- *char-operation*

A *char-operation* is: R7-32
*char-primary* // *char-primary* [ // *char-primary* ]...

A *char-primary* is one of: R7-33
- char-*constant*
- *char-variable*
- *char-function-reference*
- ( *char-expr* )


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

A *char-variable* is one of: R7-34

- char-scalar-variable-*name*
- char-array-variable-*name*
- char-*array-element-designator*
- *substring-designator*

The result of a character concatenation operation $x_1$ `//` $x_2$ 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. For example, both the expression `('AB'//'CDE')//'F'` and the expression `'AB'//('CDE'//'F')` evaluate to the string `'ABCDEF'`.

**Table 5. Interpretation of the character operator //.**

| Operator | Representing | Usage | Interpretation |
|---|---|---|---|
| `//` | Concatenation | $x_1$ `//` $x_2$ | Concatenate $x_1$ with $x_2$ |

A program may evaluate only as much of a character operation as required by the context in which the expression appears. For example, the statements

```
CHARACTER*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`.

If `A`, `B`, `C`, `X`, and `Y` are character variables, the following are valid examples of character expressions:

```
A
B // C
X // Y // 'Alpha'
```

## 7.8 Relational Expressions

A *relational expression* (R7-35) is an operation that tests whether two operands are in a specified relation to one another. A relational expression is a form of expression (R7-1).

A *relational-expr* is: R7-35

*rel-operation*


Copyright © 1989–1994 Thinking Machines Corporation

A *rel-operation* is one of: R7-36

- *num-expr rel-op num-expr*
- *char-expr rel-op char-expr*

A *rel-op* is one of: R7-37

- `.EQ.` • `==` ! equal to
- `.NE.` • `/=` ! not equal to
- `.LT.` • `<` ! less than
- `.LE.` • `<=` ! less than or equal to
- `.GT.` • `>` ! greater than
- `.GE.` • `>=` ! greater than or equal to

**NOTE:** CM Fortran also accepts the operator <> meaning "not equal to."

The first form of relational operation (R7-36) is a *numeric relational operation* involving only numeric expressions as operands; the operands are compared mathematically. The second form of relational operation is a *character relational operation*, involving only character expressions as operands; the operands are compared lexicographically. The *relational operators* (R7-37) each have two equivalent forms, as shown above (and in Table 6). All relational operators have equal precedence.

**Table 6. Interpretation of the relational operators.**

| Operator | Representing | Usage | Interpretation |
|---|---|---|---|
| `.LT.` | Less Than | $x_1$ `.LT.` $x_2$ | $x_1$ less than $x_2$ |
| `<` | Less Than | $x_1$ `<` $x2$ | $x_1$ 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 $x_2$ |

The permitted data types for operands of the relational operators are specified in Table 2, page 86. Note that, as shown in the table, a relational operator must not be used to compare the value of an expression of a numeric type with one of type character or logical. Also, two logical operands must not be compared, and a complex operand can be compared only for equality or inequality with another numeric operand.


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

fA relational operation is interpreted as having the logical value true if the values of the operands satisfy the relation specified by the operator, and false otherwise.

For a character relational operation $x_1$ *rel-op* $x_2$, the operands are compared one character at a time in order, beginning with the first character of each 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 every character of $x_1$ is the same as the character in the corresponding position of $x_2$, then $x_1$ is equal to $x_2$. Otherwise, let $c_1$ and $c_2$ be the respective characters in $x_1$ and $x_2$ at which the character operands differ. Then $x_1$ is less than $x_2$ if $c_1$ precedes $c_2$ in the ASCII collating sequence, and $x_1$ is greater than $x_2$ if $c_1$ follows $c_2$ in the collating sequence.

Two relational operations are *relationally equivalent* if their logical values are equal for all possible values of their primaries. Code generated for a relational operation may evaluate any expression that is relationally equivalent. For example, the following expressions are relationally equivalent:

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

Other examples of relational expressions:

```
A
B .EQ. C
STRING > 'AAAA'
X > 3.14159
A + B .NE. C
```

## 7.9 Logical Expressions

A *logical expression* (R7-38) is an expression (R7-1) that expresses a logical computation. Evaluation of a logical expression produces a result of type logical.

A *logical-expr* is one of: R7-38

- *logical-primary*
- *logical-operation*

A *logical-operation* is one of: R7-39

- *log-u-op logical-primary [ log-b-op logical-primary ]...*
- *logical-primary log-b-op logical-primary [ log-b-op logical-primary ]...*


Copyright © 1989–1994 Thinking Machines Corporation

A *logical-primary* is one of: R7-40

- logical-*constant*
- *logical-variable*
- logical-*array-constructor*
- logical-*function-reference*
- *relational-expr*
- ( *logical-expr* )

A *logical-variable* is one of: R7-41

- logical-scalar-variable-*name*
- logical-array-variable-*name*
- logical-*array-element-designator*
- logical-*array-section*

A *log-u-op* is: R7-42

```
.NOT.          ! logical negation
```

A *log-b-op* is one of: R7-43

- `.AND.` ! logical conjunction
- `.OR.` ! logical inclusive disjunction
- `.XOR.` ! logical nonequivalence
- `.EQV.` ! logical equivalence
- `.NEQV.` ! logical nonequivalence



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

The permitted data types for operands of the logical operators are specified in Table 2. The logical operators and their interpretation in an expression are given in Table 7.

**Table 7. Interpretation of the logical operators.**

| Operator | Representing | Usage | 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 |
| .XOR. | Logical Exclusive Disjunction | $x_1$ .XOR. $x_2$ | True if one of $x_1$ and $x_2$ is true and the other is false |
| .EQV. | Logical Equivalence | $x_1$ .EQV. $x_2$ | True if $x_1$ and $x_2$ are either both true or both false |
| .NEQV. | Logical Non-equivalence | $x_1$ .NEQV. $x_2$ | True if one of $x_1$ and $x_2$ is true and the other is false |

The values of logical operations for all possible values of their primaries are shown in Table 8.

**Table 8. Results of operations involving logical 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 |

Two logical expressions are *logically equivalent* if their values are equal for all possible values of their primaries. Code generated for a logical operation may evaluate any other expression that is logically equivalent, provided that the integrity of parentheses is not violated. For example, the following expressions are logically equivalent:

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


Copyright © 1989–1994 Thinking Machines Corporation

Other examples of logical expressions:

```
B
.NOT. B
.TRUE. .AND. 3+5 > 2
E .OR .F
1>2 .EQV. 2<1
A .AND. B .OR. .NOT. C
```

## 7.10 Bitwise Logical Expressions

As an extension to Fortran 90, CM Fortran provides six bitwise logical operators, which accept operands of type integer and return values of type integer. These operators are:

`.AND.` Performs the bitwise logical conjunction of two integer values.

`.OR.` Performs the bitwise logical inclusive disjunction of two integer values.

`.XOR.` Performs the bitwise logical exclusive disjunction of two integer values.

`.NOT.` Performs the bitwise logical complement of an integer value.

`.NEQV.` Performs the bitwise logical exclusive disjunction of two integer values.

`.EQV.` Performs `NOT`(the bitwise logical exclusive disjunction of two integer values).

These operators correspond to the bit-level intrinsic functions (described in Chapter 17) as follows:

| **Operator** | **Function** |
|---|---|
| `.AND.` | `IAND` |
| `.OR.` | `IOR` |
| `.XOR.` | `IEOR` |
| `.NOT.` | `NOT` |
| `.NEQV.` | `IEOR` |
| `.EQV.` | `NOT(IEOR)` |

Their precedence is illustrated in Table 3.


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

## 7.11 Constant Expressions

A *constant expression* is an expression (R7-1) with a constant value that can be evaluated at compile-time. Each primary (R7-2) of a constant expression must be one of the following:

- A constant (literal or named),
- An array constructor in which each element is a constant expression,
- A reference to an array inquiry intrinsic function in which each argument is a constant expression or a variable, but *not* an inquiry to determine:
  - the upper bound of an array variable in a dimension in which the bound is not a constant,
  - the upper bound of an assumed-shape array variable, or
  - the length of an assumed-length character variable.

  (Section 17.10 discusses array inquiry functions, and Section 17.3 discusses the character string length inquiry function `LEN`.)
- A substring designator that references a constant parent string and in which the substring starting point and ending point expressions are constant expressions, or
- A constant expression enclosed in parentheses.

A constant expression is either *numeric*, *character*, or *logical* depending on its type. A numeric constant expression can be *integer*, *real*, or *complex*, depending on its type. Numeric expressions are either single-precision or double-precision. The following are examples of constant expressions:

```
3                       ! integer constant expression
-3+4                    !   ditto
SIZE(A)                 !   ditto (A is explicit-shape array)
'abcd'                  ! character constant expression
'AB' //'CD'             !   ditto
('AB' //'CD') //'EF'    !   ditto
```


Copyright © 1989–1994 Thinking Machines Corporation

## 7.12 Specification Expressions

A specification expression (R7-44) is a scalar integer expression with a value that is defined on entry to a procedure. Specification expressions are useful in declaring the bounds of automatic and adjustable arrays.

A *specification-expr* is: R7-44
scalar-integer-*expr*

- A *specification-expr* is a restricted form of scalar integer expression in which each primary is one of:
  - a constant (literal or named)
  - the name of a dummy variable
  - the name of a variable in common
  - a reference to an array inquiry intrinsic function, or
  - a specification expression enclosed in parentheses

If the specification expression includes a reference to an inquiry function for an array bound, the array bound must be specified using a specification expression in the same *specification-part*.

Given the code fragment:

```
SUBROUTINE DOIT(X,C)
COMMON /GLOBALS/ N
REAL X(:)
REAL TX( UBOUND(X,1)+1 )
INTEGER MATRIX( N+1, N+1 )
CHARACTER*(*) C, CT(LEN(C))
...
```

the following are examples of specification expressions

```
UBOUND(X,1)+1
N+1
LEN(C)
```

`N` is a common variable, `X` is an assumed-shape dummy array, and `C` is an assumed-length character dummy.


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