6.1 Expression Evaluation
The world of expressions

- In the absence of side effects, expressions are relatively simple and nice objects.

- Functional programming languages rely on expressions as their “main” programming mechanisms.

- Some APL programmers liked to write one-line programs “partially motivated by the desire to stay in the more orderly world of expressions” (J. Backus CACM Aug. ‘78)
Notation

• A language may specify that function calls (operator invocations) use prefix, infix, or postfix notation.

• Infix: Most imperative languages

• Prefix: Lisp

• Postfix: Not very popular
**Precedence and Associativity**

- Useful to simplify notation and avoid too many parenthesis.
- Precedence is only relevant for infix notation.
- Some languages (APL, PL360) do not use precedence.
- For example, in PL360 the statement \( R1 := R2 + R3 \ast R4 \) was translated into (IBM 360 two address instruction set):
  \[ R1 = R2; R1 += R3; R1 *= R4; \]
- Precedence can be confusing because of the large number of operator in some languages and because different languages may have different precedence rules.
- Associativity rules are more uniform. Association is usually left-to-right except in a few cases such as \( ** \) in Fortran and assignment in C where association is right-to-left.
• In Fortran the compiler evaluates expressions in any way that provides the same interpretation. That is the compiler can use any expression that is mathematically equivalent to the original one (does not have to be computationally equivalent).

• Also, a function within a statement is not permitted to change any entity in the same statement.

• Reordering can be precluded using parenthesis.

• Usually, the operands of a binary operation can be evaluated in any order. Although Java requires left-to-right evaluation.

• The order of evaluation can affect performance.
Initialization

- Different rules in different languages/implementation regarding default initialization.

- Only safe way is to always explicitly initialize.

- Determining at compile time that a variable will always be initialized when is used is not possible in general, but in some cases (straight-line code) and some languages (Java with its definitely assigned rules) it is possible to do it.
A Brief Introduction to Fortran 90
Data Types and Kinds

Data types

• Intrisic data types (INTEGER, REAL,LOGICAL)

• derived data types (“structures” or “records” in other languages)

kind parameter (or simply kind)

• An integer that further specifies intrinsic data types (REAL(4), REAL(8))

• Literal constants (or simply literals) are specified as to kind by appending an underscore (1.5_4, 1.5_8)

• Vary from machine to machine
IMPLICIT none

When `IMPLICIT NONE` is specified, all variables have to be declared explicitly.
Examples

INTEGER, PARAMETER :: I4B = SELECTED_INT_KIND(9)
INTEGER, PARAMETER :: SP = KIND(1.0)
INTEGER, PARAMETER :: DP = KIND(1.0D0)

...  
INTEGER(I4B)  i,j,k
INTEGER m,n,p
REAL(SP)  x,y
REAL w,z
REAL(SP) :: t,u,v
READ(SP), DIMENSION(100,200) :: barr
REAL(SP) :: carr(500)
COMPLEX(KIND=SP) :: CTEMP(:)
COMPLEX(DP) :: HPCT, AA, BB(20)
Array Shapes and Sizes

The *shape* of an array refers to both its dimensionality (called its *rank*), and the length of each dimension (called the *extents*).

The F90 *intrinsic function* `shape` returns a one dimensional array (a rank-one array) whose elements are the extents along each dimension.

- `shape(barr)` returns the vector (100,200)

The *size* of an array is its total number of elements,

- The intrinsic `size(barr)` would return 20000.

The extent of each dimension can also be computed by using additional parameters.

- `size(barr,1)` returns 100
- `size(barr,2)` returns 200.
Memory Management

Within subprograms (that is, subroutines and functions), one can have

• automatic arrays that come into existence each time the subprogram is entered (and disappear when the program is exited).

• Example

  SUBROUTINE dosomething(j,k)
  REAL, DIMENSION(2*j,k**2) :: carr
Finer control on when an array is created or destroyed can be achieved by declaring *allocatable* arrays

- `REAL, DIMENSION(:,::), ALLOCATABLE :: darr`

  ```
  ...
  allocate(darr(10,20))
  ...
  deallocate(darr)
  ...
  allocate(darr(100,200))
  ...
  deallocate(darr)
  ```
• Yet finer control is achieved by the use of pointers.

• Like an allocatable array, a pointer can be allocated.

• However, it an also be *pointer associated* with a *target* that already exists under another name.

• `REAL, DIMENSION(:), POINTER :: parr`
`REAL, DIMENSION(100), TARGET :: earr`

```fortran
...  
parr => earr  
...  
nullify(parr)  
allocate(parr(500))  
...  
deallocate(parr)
```
Fortran 90 Intrinsic Procedures

aint(a,kind)  Truncate to integer value, return as a real kind
anint(a,kind) Nearest whole number, return as a real kind.
real(a,kind)  Convert to real kind

ceiling(a)  Convert to integer, truncating towards more positive
floor(a)

all(mask,dim) returns true if all elements of mask are true
any(mask,dim) Returns true if any of the elements of mask are true
count(mask,dim) counts the true elements in mask
minval(array,dim,mask) Minimum value of the array elements
maxval(array, \textit{dim}, \textit{mask})

product(array, \textit{dim}, \textit{mask})

sum(array, \textit{dim}, \textit{mask})

\begin{align*}
\text{myarray} &= \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 1 & 12 \end{bmatrix} \\
\text{sum(myarray, dim=1)} &= (15, 18, 21, 24) \\
\text{sum(myarray, dim=2)} &= (10, 26, 42) 
\end{align*}

size(array, \textit{dim})

maxloc(array, \textit{mask})

minloc(array, \textit{mask})

dot\_product(vecta, vectb)

dot\_product(vecta, vectb)

matmul(mata, matb)
Procedure Interfaces

When a procedure is referenced (called) from within a program or subprogram, the program unit must be told the procedure’s interface, that is, its calling sequence.

• INTERFACE

  SUBROUTINE caldat(julian,mm,id,iyyy)
  INTEGER, INTENT(IN) :: julian
  INTEGER, INTENT(OUT) :: MM,ID,IYYY
  END SUBROUTINE caldat

END INTERFACE
**Triplet notation**

Sections of arrays are identified in Fortran 90 using triplets of the form \( l : u : s \). A triplet represents the sequence of subscripts

\[
\begin{align*}
  l, & \; l+s, \; l+2s, \ldots, l+ms
\end{align*}
\]

where \( m \) is the smallest number such that

\[
\begin{align*}
  l+(m+1)s & > u \quad (\text{if } s \geq l) \\
  l+(m+1)s & < u \quad (\text{if } s \leq l)
\end{align*}
\]

For example, the section \( A(3:5,2,1:2) \) of an array \( A \) is the array of shape \((3,2)\):

\[
\begin{align*}
  A(3,2,1) & \quad A(3,2,2) \\
  A(4,2,1) & \quad A(4,2,2) \\
  A(5,2,1) & \quad A(5,2,2)
\end{align*}
\]

If \( l \) is omitted, the lower bound for the array is assumed. If \( u \) is omitted, the upper bound is assumed. If \( s \) is omitted, 1 is assumed. The stride \( s \) cannot be 0.
Expressions in Fortran 90 may contain array sections, specified using triplets, or complete arrays identified by the name of the array without any subscripts.

For example, consider the arrays a, b and c declared as follows:
```fortran
dimension a(100,100) b(100,100),c(100,100)
```
The statement
```fortran
c = a + b
```
assigns to matrix c the element-by-element sum of matrices a and b.

Also,
```fortran
a(1:100, 2) = 0
```
assigns 0 to the second column of a. An identical function is performed by the following three statements.
```fortran
a(:100,2) = 0
a(1:,2) = 0
a(:,2)  = 0
```

Another example is
```fortran
a(51:100,4) = b(1:50,4) * c(30,31:80)
a(51:100,4) = a(50:99,4) + 1
```

- The rank of an array is the number of dimensions.
• The *shape* of an array is determined by its rank and its extent in each dimension.

• All the objects in an expression or assignment statement must be *conformable*. Two arrays are conformable if they have the same shape. A scalar is conformable with any array.

• Any intrinsic operation defined for scalar objects may be applied to conformable objects. Such operations are performed element-by-element to produce a resultant array conformable with the array operands.

• The masked array assignment is used to perform selective assignment to arrays. For example, in the statement
  
  ```
  where(temp>0)temp = temp - reduce_temp
  ```
  
  only those elements in the array `temp` which are $> 0$ will be decreased by the value `reduce_temp`.

In the following compound statement,
  ```
  where(pressure<=0)
  pressure = pressure + inc_pressure
  ```
temp = temp - 5.0
elsewhere
    raining = .true.
end where

the array pressure in modified only where it is <= 1. Also, the array temp is modified in the corresponding locations (i.e. in the same locations as pressure). Finally, the array raining is assigned .true. only in the locations that correspond to those element of pressure which are > 1.

• The mask of the where statement is like another operator on the right-hand side of all the assignment statements in the body of the where statement and therefore has to be conformable to the right-hand side expression and to the array on the left-hand side.

• There are a collection of intrinsic functions designed to operate on arrays. These will be described as needed.