Table 5.3 Mixed-Mode Evaluation

Mixed-mode expression	*Evaluation*	*Result type*
integer *op* real	Convert the integer to the corresponding real value and evaluate the expression.	Real
integer *op* double precision	Convert the integer to the corresponding double precision value and evaluate the expression.	Double precision
real *op* double precision	Extend the real to a double precision value (by adding zeros) and evaluate the expression.	Double precision

5.8 CONTROL OF EXECUTION

Control of execution means the transfer of execution from one point to another in the same program, depending on the conditions of certain variables. This may involve a *forward jump* thus skipping a block of statements, or a *backward jump* thus repeating the execution of a block of statements. This is known as *conditional execution* of statements. Examples of such conditional execution are:

1. If the value is negative, skip the following four statements.
2. If the item is the last one, go to the end.
3. Execute the following ten lines 100 times.
4. Evaluate the following statement until a given condition is satisfied.

FORTRAN contains two central structures which could be used to implement such conditional execution of statements. They are

1. IF-ELSE structure
2. DO-WHILE structure

Block IF-ELSE Structure

The block IF-ELSE structure (also known as *selection* structure) consists of a logical expression that tests for a condition or a relation followed by two alternative paths for the execution to follow. Depending on the test results, one of the paths is executed and the other is skipped. This is

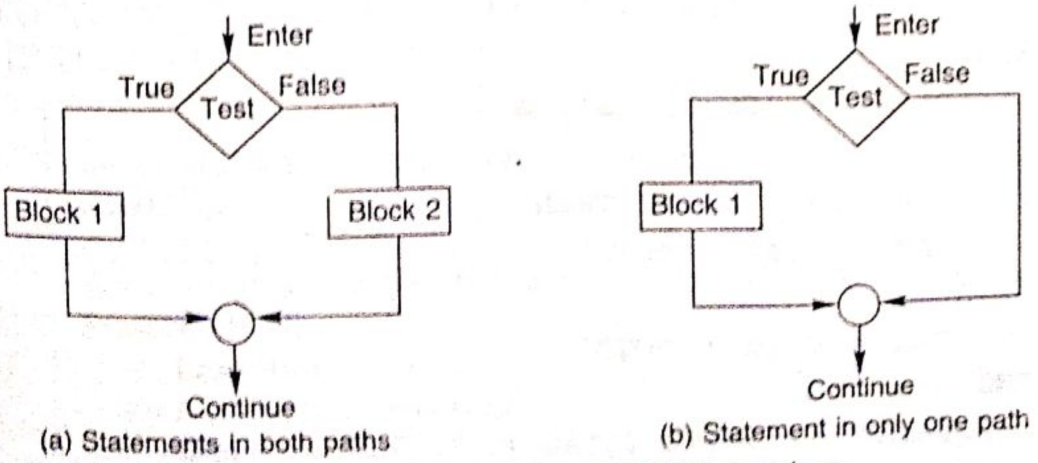

Fig. 5.1 Flow chart of IF-ELSE structure

illustrated in Fig. 5.1.

The FORTRAN statement to code a block IF-ELSE structure takes the form:

```
IF (logical expression) THEN          IF block
    statement-block 1
ELSE                                  ELSE block
    statement-block 2
END IF
```

The statement blocks may contain zero or more statements. If the logical expression is true, the program executes statement-block 1 and then goes to the statement next to the END IF statement; if the logical expression is false, the program executes statement-block 2 (skipping statement-block 1) and then goes to the statement next to the END IF.

Relational Expressions

Relational expressions are meant for comparing the values of two arithmetic expressions and have logical values .TRUE. or .FALSE. as results. Arithmetic expressions may contain single variable, simple constant, intrinsic function, or a complex expression. In numerical computing, we often want our programs to test for certain relationships and make decisions based on the outcomes. We may use the *relational operators* given in Table 5.4 for comparing the expressions.

Table 5.4 Relational operators

Operator	*Meaning*
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Examples of rational operators are

```
1. IF(X .LT. Y) THEN
        PRINT * 'Small is', X
   ELSE
        PRINT * 'Small is', Y
   END IF
2. IF(TOTAL .GT. 1000) THEN
        TAX = 0.15 * TOTAL
   ELSE
        TAX = 0.10 * TOTAL
   END IF
        PRINT * 'GRAND_TOTAL = ', TOTAL + TAX
```

```
3. IF(C - D .GE. A - B) THEN
       X = C - D
   ELSE
       X = A - B
   END IF
```

When arithmetic expressions are used along with the relational operators, arithmetic expressions are evaluated first and then the results are compared.

Logical Expressions

In some cases, we may need to make more than one comparison. It is possible to combine two relational expressions using the following logical operators:

`.AND.`	Both relations are true
`.OR.`	One or both of the relations are true
`.NOT.`	Opposite is true

Such expressions are known as *logical expressions.*
Examples of logical expressions are:

```
1. IF(SUM .GT. 100 .OR. N .GT. 20) THEN
           ...
           ...
   ELSE
           ...
           ...
   END IF
2. IF(AGE .LT. 30 .AND. DEGREE .EQ. 'ME') THEN
           ...
           ...
   ELSE
           ...
           ...
   END IF
```

FORTRAN permits nesting of IF-ELSE blocks. That is, we can place an IF-THEN-ELSE code within an IF block or ELSE block.

Warning!
Be careful when comparing real values. They are never exact!

We may also use the following relational operators in FORTRAN 90.

<	Less than
<=	Less than or equal to
= =	Equal to

```
/= Not equal to
>= Greater than or equal to
 > Greater than
```

DO-WHILE Structure

The DO-WHILE structure (also known as *looping structure*) performs a set of operations repeatedly while a certain condition is true. When the condition is not true, the repetition ceases. This kind of structure is implemented in FORTRAN by the DO statement. The general format of DO statement is:

```
DO n i = e1, e2, e3
        ...
        ...  <-------------- Body of the loop
        ...
n CONTINUE
```

where

n	number of the last statement in the loop
i	loop control variable
e_1	initial value of the control variable
e_2	final value of the control variable
e_3	increment value.

The control variable i may be a real or integer variable. The parameters e_1, e_2, and e_3 may be real or integer variables (or expressions or constants).

The default value of e_3 is 1. The logic of DO loop is as follows:

1. initialise the loop control variable to the initial value e_1
2. test to see if the value of loop control variable is less than or equal to the final value e_2. If it is true, continue the loop; otherwise exit the loop
3. execute the body of the loop
4. increment the loop control variable by e_3
5. go back to step 2 (beginning of the loop)

This can be written in pseudocode form as follows:

```
i = e1
DO WHILE i <= e2
    execute statements
    i = i + e3
END DO
```

Figure 5.2 shows a flow chart showing the execution of the DO structure. The number of times the loop is executed (unless terminated by an EXIT statement) is given by the formula

$$m = \left\lfloor \frac{e_2 - e_1 + e_3}{e_3} \right\rfloor$$

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

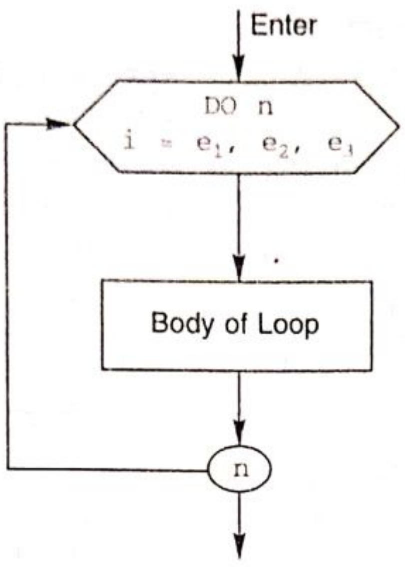

Fig. 5.2 Flow chart for the DO loop

Examples of DO loop are

```
1.      DO 10 P = X/Y, 75, Z/10.0
        ...
        ...
     10 CONTINUE
2.      DO 20 I = -4, 10, 0.25
        ...
        ...
     20 CONTINUE
3.      DO 30 N = 2, 20
        ...
        ...
     30 CONTINUE
4.      DO 40 J = 1, 100
        ...
        ...
        IF (...) GOTO 50   (Exit from the loop)
     40 CONTINUE
     50 ...
```

Warning !

Avoid the use of real variables for DO loop parameters. They cause roundoff errors and, therefore, cannot always guarantee the correct number of loop executions.

A DO loop can contain DO loops within its range. This is known as *nesting*. When nesting DO loops, the inner loop must be entirely contained within the range of the outer loop.

Examples of nesting DO loops are

```
1.      DO 200 I = 1, 10                  outer loop
            DO 100 J = 1, 5
            PRINT *, I * J                    inner loop
  100       CONTINUE
  200   CONTINUE

2.      DO 500 I = 1, 10
            DO 200 J = 1, 10              outer loop
            ...
            ...
  200       CONTINUE
        ...                                   inner loops
        ...
            DO 300 K = 1,5
            ...
            ...
  300       CONTINUE
  500   CONTINUE
```

The general form of DO structure in FORTRAN 90 is:

```
DO   loop control
     block of statements
END DO
```

This is implemented in two forms:

Form 1

```
DO i = e₁, e₂, e₃
      ...
      ...
END DO
```

Form 2

```
1.  DO
         ...
         ...
         IF (...) EXIT
         ...
    END DO
    ...                                   Leave the loop
2.  DO
         ...
         ...
         IF (...) CYCLE                   Go to the beginning
         ...
         ...
    END DO
```

5.9 SUBPROGRAMS

One of the features of any modern programming language is the provision for *subprograms*. A subprogram is a separate program unit that can be called into operation by other programs. Subprograms are heavily used in numerical computing for tasks such as evaluation of a function, matrix multiplication, sorting, reading a table of values, printing a report, etc.

The concept of subprograms allows us to break a complex problem into subtasks so that we may develop subprograms and later integrate them into a single program known as *driver* or *main* program. These subprograms can be independently designed, coded, and tested. Subprograms are usually called *modules* and the programming approach using modules is called *modular programming*.

✗ FORTRAN supports two kinds of subprograms, namely, *functions* and *subroutines*. A function subprogram returns a single value to the calling program while a subroutine subprogram can compute and return several values.

Function Subprograms

A function subprogram (or simply a function) is an independent program unit written to compute and return a single value. It takes the following form:

```
type  FUNCTION name (arguments)
      Declaration of argument types
      ...
      ...   <------------------------ Execution statements
      ...
      name = expression
      RETURN
      END
```

where `type` specifies the type of the function value that is being returned and `arguments` are *dummy* variables that must be declared for their type inside the function. They may vary in number from zero to many. There should be at least one statement of the form

`name = expression`

which assigns a value of appropriate type to the function name, which is in turn returned to the calling prbgram.

A function can be called as follows:

`variable = name (arguments)`

When the function is called, the values of the arguments in the calling statement are assigned to the corresponding arguments in the function header. The arguments, therefore, must agree in order, number and type. An argument may be a variable name, an array name, or a subprogram name. Example:

```
PROGRAM MAIN
REAL A, B, R, MUL  ←——— MUL declared in main
READ * A, B
R = MUL(A, B)      ←——— Calling MUL
PRINT * , R
END
_______________________________________
REAL FUNCTION MUL(X, Y) ←——— MUL defined
REAL X, Y
MUL = X * Y
RETURN
END
```

When an array is passed as an argument, then its corresponding dummy argument should be an array variable and its size must be declared properly. Note that a function may be called and used in an expression, like any other variable. Example:

```
R = A * MUL (A, B)
```

Subroutine Subprogram

A subroutine, unlike a function which always returns only one value, can return many values (or no values). Therefore, we use a subroutine when either several values are to be computed and returned or no values are to be returned (such as printing the values of some variables). The general structure of a subroutine is:

```
SUBROUTINE name (arguments)
Declaration of arguments
...
...      ←——— Execution statements
...
RETURN
END
```

where *name* is the subroutine name and *arguments* are *dummy* variables that must be declared for their type. When subroutine has no arguments, the parentheses are omitted (note that in case of function, parentheses ~~are necessary even if there are no arguments~~). The outputs of subroutine are returned to the calling program by means of the arguments.

A subroutine can be invoked using the CALL statement as follows:

```
CALL name (arguments)
        or
CALL name
```

The actual arguments in the calling statement must agree in a one-to-one manner with the order and type of the arguments in the subroutine. Example:

```
PROGRAM MAIN
REAL A, B, R
READ * , A , B
CALL MUL (A, B, R)
PRINT *, R
END
```

```
SUBROUTINE MUL(X, Y, XY)
REAL X, Y, XY
XY = X * Y
RETURN
END
```

The calling program assigns the values of A and B to the variables X and Y in the subroutine which in turn assigns the value of XY (computed in the subroutine) to the variable R. Compare this with the function subprogram.

Note that the variables that are not passed as arguments may be passed to the subroutine using a COMMON statement.

FORTRAN 90 greatly extends the power of function subprograms by allowing the result to be an array or structure. Function subprograms are designed as follows:

```
FUNCTION name(arguments) RESULT (result -
                                  variable)
Declaration of arguments and result-variable
...
...
result-variable = expression
END FUNCTION name
```

Instead of function name, the result-variable is assigned the value that is to be returned to the calling function. The result-variable is a variable name that has been placed like a function argument with the RESULT keyword, immediately after the function name. Both the arguments and the result-variable are declared for their types.

Note that, in FORTRAN 90, all programs and subprograms use the name of the program or subprogram in the END statement as follows:

```
END FUNCTION F
END SUBROUTINE SWAP
END PROGRAM SORT
```

FORTRAN 90 also includes features such as optional arguments, keyword-identified arguments and array sizes which are very powerful compared to FORTRAN 77. These features must be used wherever possible.

FORTRAN allows us to write out a formula for a function and define it using the assignment statement inside the program itself (instead of using an "external" function subprogram). Since such functions are "one-line" functions, they are called *statement functions*. A statement function is defined as follows:

Function-name (*arguments*) = expression

where *expression* is the FORTRAN expression of the formula (or function) to be evaluated and *arguments* is a list of variables used in the expression. The arguments are simple integer or real variables. Examples:

```
AREA  (R)             = 3.1416 * R * R
VALUE  (P,  R)        = P  *  (1.0  +  R)  **  N
POLY  (X,  Y,  M,  N) = X  **  M  +  Y  **  N
```

A variable which appears in the expression but is not defined as an argument is called the *parameter* of the function. Values of such variables should be defined before using the function.

The function can be used in any subsequent lines of the program by writing the name of the function with actual arguments, like

```
CIRCLE   =  AREA(X)
FVALUE   =  VALUE(AMOUNT,  INTEREST)
POLY1    =  POLY  (A,  2,  B,  2)
RING     =  AREA(X1)  -  AREA(X2)
```

Note that the functions can be used on the right side like any other variables. The actual arguments may be variables or constants (or even expressions). However, they must agree in number order and type with the dummy arguments in the function definition statement.

A statement function may use other statement functions if they are defined before it. Like function subprograms, the statement functions must be declared for their type in the program and defined after all declarations, but before the first executable statement.

5.10 INTRINSIC FUNCTIONS

In numerical computing, we use mathematical functions like logarithm, square root, absolute value, sine, etc., very frequently. FORTRAN supports a library of such functions which can be invoked in our programs. Since these functions are part of FORTRAN, they are also called *intrinsic* or *built-in functions*. An intrinsic function can be invoked by simply typing the name of the function followed by the arguments enclosed in parentheses. Example:

```
ABS(X)     COS(THETA)     SQRT(X  *  X  +  Y  *  Y)
```

The most commonly used intrinsic mathematical functions are summarised in Table 5.5. When using any of these functions, it is a good practice to declare them using the INTRINSIC statement in the declaration section.