5 REPETITION

While writing a program, it may be necessary to execute a statement or a group of statements repeatedly. Repetition is supported in FORTRAN through two repetition constructs, namely, the **DO** and the **WHILE** constructs. A repetition construct is also known as a *loop*.

In a repetition construct, a group of statements, which are executed repeatedly, is called the *loop body*. A single execution of the loop is called an *iteration*. Every repetition construct must *terminate* after a *finite* number of iterations. The termination of the loop is decided through what is known as the *termination condition*. A decision is made whether to execute the loop for another iteration through the termination condition. In the case of a **DO** loop, the number of iterations is known before the loop is executed; the termination condition checks whether this number of iterations have been executed. In the case of a **WHILE** loop, such a decision is made in every iteration.

Repetition constructs are very useful and extensively used in solving a significant number of programming problems. Let us consider the following example as an illustration of such constructs.

Example : *Average Computation: Assume that we were asked to write a FORTRAN program that reads the grades of 8 students in an exam. The program is to compute and print the average of the grades. Without repetition, the following program may be considered as a solution:*

Solution:

```
      REAL X1, X2, X3, X4, X5, X6, X7, X8
      REAL SUM, AVG
      READ*, X1
      READ*, X2
      READ*, X3
      READ*, X4
      READ*, X5
      READ*, X6
      READ*, X7
      READ*, X8
      SUM = X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8
      AVG = SUM / 8.0
      PRINT*, AVG
      END
```

The variable SUM is a real variable in which we store the summation of the grades. The statements are considerably long for just 8 students. Imagine the size of such statements when the number of students is 100. It is highly inefficient to use 100 different variable names.

From the example above, let us try to extract the statements where repetition occurs. The reading and assignment statements are clearly such statements. We can do the reading and addition in these statements, individually, for each grade. The following repetitive segment can be used instead of the long read and assignment statements :

```
      SUM = 0
      REPEAT THE FOLLOWING STATEMENTS 8 TIMES
          READ*, X
          SUM = SUM + X
```

In each iteration, one grade is read and then added to the previous grades. In the first iteration, however, there are no previous grades. Therefore, SUM is initialized to zero, meaning that the summation of the grades is zero, before any grade is read.

This repetitive solution is more efficient since it can be used for any number of students. By reading the number of students N, the repetition construct above, can be changed, to find the sum of the grades of N students, as follows :

```
      SUM = 0
      READ*, N
      REPEAT THE FOLLOWING STATEMENTS N TIMES
          READ*, X
          SUM = SUM + X
```

The repetition construct above is not written in the FORTRAN language. To implement this construct in FORTRAN, we can use two types of loops: the **DO** Loop and the **WHILE** loop.

5.1 The DO Loop

One very basic feature of the **DO** loop repetitive construct is that the number of iterations (the number of times the loop is executed) is known (computed) before the loop execution begins. The general form of the **DO** loop is:

```
      DO N index = initial, limit, increment
          BLOCK OF FORTRAN STATEMENTS
N     CONTINUE
```

The **CONTINUE** statement indicates the end of the **DO** loop.

The number of times (iterations) the loop is executed is computed as follows :

$$\text{Number of times a Do loop is Executed} = \left\lceil \frac{\text{limit} - \text{initial}}{\text{increment}} \right\rceil + 1$$

The detailed logic of the **DO** loop is as follows:

- If the *increment* is positive, the value of the *initial* must be less than or equal to the value of the *limit*. If the *increment* is negative, the value of the *initial* must be greater than or equal to the value of the *limit*. Otherwise, the loop will not be executed. If the values of the *initial* and the *limit* are equal, the loop executes only once.
- In the first iteration, the *index* of the loop has the value of *initial* .
- Once the **CONTINUE** statement is reached, the *index* is increased or decreased by the *increment* and the execution of the next iteration starts. Before each

iteration, the *index* is checked to see if it has reached the *limit*. If the *index* reaches the *limit*, the loop iterations stop. Otherwise, the next iteration begins.

Consider the following example as an illustration of the **DO** loop :

```
      DO 15 K = 1, 5, 2
          PRINT*, K
15    CONTINUE
```

The loop above is executed $\left\lceil \frac{5-1}{2} \right\rceil + 1 = 3$ times. Thus, the values index K takes during the execution of the loop are 1, 3, and 5. Note that the value of K increments by 2 in each iteration. In the beginning, we make sure that the initial is less than the limit since the value of the increment is positive. The execution of the loop begins and the value of K, which is **1**, is printed. The **CONTINUE** statement returns the control to the **DO** statement and the execution of the loop takes place for the second time with the value of K as **3**. This continues for the third time with K as **5**. Once this iteration is over, the control goes back and the *index* K gets incremented again to **7**, which is more than the *limit*. The execution of the loop stops and control transfers to the statement following the **CONTINUE** statement. Note that the value of K outside the loop is 7.

The following **rules** apply to **DO** loops:

- The *index* of a **DO** loop must be a variable of either **INTEGER** or **REAL** types.
- The parameters of the loop, namely, *initial*, *limit*, and *increment* can be expressions of either **INTEGER** or **REAL** types. Although it depends on the nature of the problem being solved, it is recommended that the type of the parameters match the type of the *index*.
- The value of the **DO** loop *index* cannot be modified inside the loop. Any attempt to modify the *index* within the loop will cause an error.
- The *increment* must not be **zero**, otherwise an error occurs.
- If the *index* is an integer variable then the values of the parameters of the **DO** loop will be truncated to integer values before execution starts.
- The value of the index after the execution of the loop is either the value that has been incremented and found to exceed the limit (for a positive increment) or the value that has been decremented and found to be less than the limit (for a negative increment).
- It is not allowed to branch into a **DO** loop. Entering the **DO** loop has to be through its **DO** statement. It is possible to branch out of a **DO** loop before all the iterations are completed. This type of branching must not be used unless necessary.
- It is possible to have a **DO** loop without the **CONTINUE** statement. The *statement number*, which is given to the **CONTINUE** statement, can be given to the last FORTRAN statement in the loop, except in the case when the last statement is either an **IF**, **GOTO**, **RETURN**, **STOP** or another **DO** statement.
- In the **DO** loop construct, in the absence of the increment, the default increment is +1 or +1.0 depending on the type of the *index*.

- In the case when the *increment* is positive but the *initial* is greater than the *limit*, a **zero-trip DO** loop occurs. That is, the loop executes zero times. The same happens when the *increment* is negative and the *initial* is less than the *limit*. Note that a zero-trip **DO** loop is not an error.
- The same continue statement number can be used in both a subprogram and the main program invoking the subprogram. This is allowed because subprograms are considered separate programs.
- The parameters of the loop are evaluated before the loop execution begins. Once evaluated, changing their values will not affect the executing of the loop. For an example, consider the following segment. Changing **DO** loop parameters inside the loop should be avoided while writing application programs.

```
      REAL X, Y
      Y = 4.0
      DO 43 X = 0.0, Y, 1.5
         PRINT*, X
         Y = Y + 1.0
         PRINT*, Y
43    CONTINUE
```

In the above loop, the value of Y which corresponds to the limit in the **DO** loop, starts with **4**. Therefore, and according to the rule we defined earlier, this loop is executed $\left\lceil \frac{4.0-0.0}{1.5} \right\rceil + 1 = 3$ times. The values of the parameters (*initial, limit,* and *increment*) are set at the beginning of the loop and they never change for any iteration of the loop. Although the value of Y changes in each iteration within the loop, the value of the limit does not change. The following examples illustrate the ideas explained above:

5.1.1 Examples on DO loops

Example 1: *Consider the following program.*

```
      DO 124 M = 1, 100, 0.5
         PRINT*, M
124      CONTINUE
      PRINT*, M
      END
```

In the above program, the value of the increment is ***0.5****. When this value is added and assigned to the index M, which is an* ***integer****, the fraction part gets truncated. This means that the increment is* ***0*** *which causes an error.*

Example 2: The Factorial: *Write a FORTRAN program that reads an integer number M. The program then computes and prints the factorial of M.*

Solution:

```
      INTEGER M, TERM, FACT
      READ*, M
      IF (M.GE.0) THEN
          FACT = 1
          TERM = M
           DO 100 M = TERM, 2, -1
              IF (TERM.GT.1) THEN
                  FACT = FACT * TERM
100         CONTINUE
           PRINT*,'FACTORIAL OF ', M, ' IS ', FACT
      ELSE
          PRINT*, 'NO FACTORIAL FOR NEGATIVES'
      ENDIF
      END
```

To compute the factorial of 3, for example, we have to perform the following multiplication: 3 * 2 * 1. Notice that the terms decrease by 1 and stop when the value reaches 1. Therefore, the header of the **DO** loop forces the repetition to stop when TERM, which represents the number of terms, reaches the value 1.

5.2 Nested DO Loops

DO loops can be nested, that is you may have a **DO** loop inside another **DO** loop. However, one must start the inner loop after starting the outer loop and end the inner loop before ending the outer loop. It is allowed to have as many levels of nesting as one wishes. The constraint here is that inner loops must finish before outer ones and the indexes of the nested loops must be different. The following section presents some examples of nested **DO** loops.

5.2.1 Example on Nested DO loops

Example 1: *Nested **DO** Loops: Consider the following program.*

```
          DO 111 M = 1, 2
              DO 122 J = 1, 6 , 2
                  PRINT*, M, J
122           CONTINUE
111       CONTINUE
          END
```

The output of the above program is:

```
1   1
1   3
1   5
2   1
2   3
2   5
```

Example 2: *The above program can be rewritten using one **CONTINUE** statement as follows:.*

```
          DO 111 M = 1, 2
              DO 111 J = 1, 6 , 2
                  PRINT*, M, J
111       CONTINUE
          END
```

Notice that both do loops has the same label number and the same ***CONTINUE*** *statement.*

Example 3: *The above program can be rewritten without any* ***CONTINUE*** *statement as follows:*

```
          DO 111 M = 1, 2
              DO 111 J = 1, 6 , 2
111               PRINT*, M, J
          END
```

Notice that the label of the do loop will be attached to the last statement in the do loop.

5.3 The WHILE Loop

The *informal* representation of the WHILE loop is as follows :

```
       WHILE condition EXECUTE THE FOLLOWING
           block of statementS.
```

In this construct, the *condition* is checked before executing the *block of statements*. The *block of statements* is executed only if the *condition*, which is a logical expression, evaluates to a *true* value. At the end of each iteration, the control returns to the beginning of the loop where the *condition* is checked again. Depending on the value of the *condition*, the decision to continue for another iteration is made. This means that the number of iterations the **WHILE** loop makes depends on the *condition* of the loop and could not always be computed before the execution of the loop starts. This is the main difference between **WHILE** and **DO** repetition constructs.

Unlike other programming languages such as PASCAL and C, standard FORTRAN does not have an explicit **WHILE** statement for repetition. Instead, it is built from the **IF** and the **GOTO** statements.

In FORTRAN, the **IF-THEN** construct is used to perform the test at the beginning of the loop. Consider an **IF** statement, which has the following structure :

```
       IF (condition) THEN
           block of statements
       ENDIF
```

If the condition is .TRUE., the *block of statements* is executed once. For the next iteration, since we need to go to the beginning of the **IF** statement, we require the **GOTO** statement. It has the following general form :

```
       GOTO statement number
```

A **GOTO** statement transfers control to the statement that has the given statement number. Using the **IF** and the **GOTO** statements, the general form of the **WHILE** loop is as follows :

```
n      IF (condition) THEN
           block of statements
           GOTO n
       ENDIF
```

n is a positive integer constant up to 5 digits and therefore, ranges from 1 to 99999. It is the label of the **IF** statement and must be placed in columns 1 through 5.

The execution of the loop starts if the *condition* evaluates to a *.TRUE.* value. Once the loop iterations begin, the *condition* must be ultimately changed to a .FALSE. value, so that the loop stops after a finite number of iterations. Otherwise, the loop never stops resulting in what is known as the *infinite* loop. In the following section, we elaborate more on the **WHILE** loop.

5.3.1 Examples on WHILE Loops

Example 1: *Computation of the Average: Write a FORTRAN program that reads the grades of 100 students in a course. The program then computes and prints the average of the grades.*

Solution:

```
      REAL X, AVG, SUM
      INTEGER K
      K = 0
      SUM = 0.0
25    IF (K.LT.100) THEN
         READ*, X
         K = K + 1
         SUM = SUM + X
         GOTO   25
      ENDIF
      AVG = SUM / K
      PRINT*, AVG
      END
```

Note that the variable K starts at 0. The value of K is incremented after the reading of a grade. The **IF** condition presents the loop from reading any new grades once the 100th grade is read. Reading the 100th grade causes K to be incremented to the value of 100 as well. Therefore, when the condition is checked in the next iteration, it becomes .FALSE. and the loop stops.

In each iteration, the value of the variable GRADE is added to the variable SUM. After the loop, the average is computed by dividing the variable SUM by the variable K.

Example 2: *The Factorial: The problem is the same as the one discussed in Example 2 of Section 5.2. In this context, however, we will solve it using a* ***WHILE*** *loop.*

Solution:

```
      INTEGER M, TERM, FACT
      READ*, M
      IF (M.GE.0) THEN
         FACT = 1
         TERM = M
3     IF (TERM.GT.1) THEN
         FACT = FACT *TERM
         TERM =TERM - 1
         GOTO 3
      ENDIF
         PRINT*,'FACTORIAL OF ', M, ' IS ', FACT
      ELSE
         PRINT*, 'NO FACTORIAL FOR NEGATIVES'
      ENDIF
      END
```

Note the similarities between both solutions. The **WHILE** loop starts from **M** (the value we would like to compute the factorial of) and the condition of the loop makes sure that the loop will only stop when TERM reaches the value 1.

Example 3: *Classification of Boxers: Write a FORTRAN program that reads the weights of boxers. Each weight is given on a separate line of input. The boxer is classified according to the following criteria: if the weight is less than or equal to 65 kilograms, the boxer is light-weight; if the weight is between 65 and 85 kilograms, the boxer is middle-weight and if the weight is more than or equal to 85, the boxer is a heavy-weight. The program prints a proper message according to this classification for a number of boxers by reading their weights repeatedly from the input. This repetitive process of reading and classification stops when a weight of* ***-1.0*** *is read.*

Solution:

```
      REAL WEIGHT
      READ*, WEIGHT
11    IF (WEIGHT.NE.-1.0) THEN
          IF (WEIGHT.LT.0.OR.WEIGHT.GE.400) THEN
              PRINT*, ' WEIGHT IS OUT OF RANGE '
          ELSEIF (WEIGHT.LE.65) THEN
              PRINT*, ' LIGHT-WEIGHT '
          ELSEIF (WEIGHT.LT.85) THEN
              PRINT*, ' MIDDLE-WEIGHT '
          ELSE
              PRINT*, ' HEAVY-WEIGHT '
          ENDIF
          READ*, WEIGHT
          GOTO 11
      ENDIF
      END
```

Note that in this example, the condition that stops the iterations of the loop depends on the **READ** statement. The execution of the loop stops when a value of **-1.0** is read. This value is called the ***end marker*** or the ***sentinel***, since it marks the end of the input. A sentinel must be chosen from outside the range of the possible input values.

5.4 Nested WHILE Loops

WHILE loops may be nested, that is you can put a **WHILE** loop inside another **WHILE** loop. However, one must start the inner loop after starting the outer loop and end the inner loop before ending the outer loop for a logically correct nesting. (The following example is equivalent to the nested **DO** loop example given earlier.)

Example: Consider the following program.

```
      M = 1
22    IF( M .LE. 2) THEN
          J = 1
11        IF (J .LE. 6) THEN
              PRINT*, M, J
              J = J + 2
              GOTO 11
          ENDIF
          M = M + 1
          GOTO 22
      ENDIF
      END
```

The output of the above program is:

```
1   1
1   3
1   5
```

```
2   1
2   3
2   5
```

There are two nested **WHILE** loops in the above program. The outer loop is controlled by the variable M. The inner loop is controlled by the variable J. For each value of the variable M, the inner loop variable J takes the values 1, 3 and 5.

5.5 Examples on DO and WHILE Loops

Example 1: *Evaluation of Series: Write a FORTRAN program that evaluates the following series to the 7th term.*

$$\sum_{i=1}^{N} 3^{i}$$

(Summation of base 3 to the powers from 1 to N. Assume N has the value 7)

Solution:

```
      INTEGER SUM
      SUM = 0
      DO 11 K = 1, 7
          SUM = SUM + 3 ** K
11    CONTINUE
      PRINT*, SUM
      END
```

Example 2: *Alternating Sequences/ Series: Alternating sequences, or series, are those which have terms alternating their signs from positive to negative. In this example, we find the sum of an alternating series.*

Question: Write a FORTRAN program that evaluates the following series to the 100th term.

1 - 3 + 5 - 7 + 9 - 11 + 13 - 15 + 17 - 19 + ...

Solution:

It is obvious that the terms differ by 2 and start at the value of 1.

```
      INTEGER SUM, TERM,NTERM
      SUM = 0
      TERM = 1
      DO 10 NTERM = 1, 100
          SUM = SUM + (-1) ** (NTERM + 1) * TERM
          TERM = TERM + 2
10    CONTINUE
      PRINT*, SUM
      END
```

Notice the summation statement inside the loop. The expression (-1) ** (NTERM + 1) is positive when NTERM equals 1, that is for the first term. Then, it becomes negative for the second term since NTERM + 1 is 3 and so on.

Example 3: *Series Summation using a WHILE loop: Question: Write a FORTRAN program which calculates the sum of the following series :*

$$\frac{1}{2}+\frac{2}{3}+\frac{3}{4}+\frac{4}{5}+\text{L}+\frac{99}{100}$$

Solution:

```
      REAL N, SUM
      N = 1
      SUM = 0
10    IF (N.LE.99) THEN
          SUM = SUM + N / (N + 1)
           N = N + 1
          GOTO 10
      ENDIF
      PRINT*, SUM
      END
```

In the above program, if N is not declared as **REAL**, the expression N/(N+1), in the summation inside the loop, will always compute to zero.

Example 4: *Conversion of a WHILE loop to a* ***DO*** *loop: Convert the following* ***WHILE*** *loop into a* ***DO*** *loop.*

```
      REAL X, AVG, SUM
      INTEGER K
      K = 0
      SUM = 0.0
25    IF (K.LT.100) THEN
          READ*, X
          K = K + 1
          SUM = SUM + X
          GOTO 25
      ENDIF
      AVG = SUM / K
      PRINT*, AVG
      END
```

In the **WHILE** loop, K starts with the value of 0, and within the loop it is incremented by 1 in each iteration. The *termination condition* is that the value of K must exceed 99. In the equivalent program using a **DO** loop, K starts at 0 and stops at 99 and gets incremented by 1 in each iteration.

Solution:

The equivalent program using a **DO** loop is as follows:

```
      REAL X, AVG, SUM
      INTEGER K
      SUM = 0.0
      DO 25 K = 0, 99, 1
          READ*, X
          SUM = SUM + X
 25   CONTINUE
      AVG = SUM / 100
      PRINT*, AVG
      END
```

An important point to note in this example is the way the average is computed. The statement that computes the average divides the summation of the grades SUM by 100. Note that the value of the K is 100 because the loop stops when the value of K exceeds 99. Keeping in mind that the increment is 1, the value of K after the loop terminates is 100. However, it is not recommended to use the value of the index outside the **DO** loop.

It is also important to note that any other parameters such as:

```
      DO 25 K = 200, 101, -1
```

would also have the same effect. Note that the variable K exits the loop with the value 100 in this case as well.

It is not always possible to convert a **WHILE** loop into a **DO** loop. As an example, consider the **WHILE** loop in the Classification of Boxers example. There, we cannot accomplish the conversion because the number of times the **WHILE** loop gets executed is not known. It depends on the number of data values before the *end marker*.

5.6 Implied Loops

Implied loops are only used in **READ** and **PRINT** statements. The implied loop is written in the following manner :

```
      READ*,(list of variables, index = initial, limit, increment)
      PRINT*,(list of expressions, index = initial, limit, increment)
```

As in the case of explicit **DO** loops, the index must be either an integer or real expression. The variables in the **READ** statement can be of any type including array elements. The expressions in the **PRINT** statement can be of any type as well. All the rules that apply to **DO** loop parameters also apply to implied loop parameters. Usage of implied loops is given in the following examples :

Example 1: *Printing values from 100 to 87: The following segment prints the integer values from 100 down to 87 in a single line.*

```
      PRINT*, (K, K = 100 , 87 , -1)
```

Output:

```
100  99   98   97   96   95   94   93   92   91   90   89   88   87
```

Notice that the increment is -1, which means that the value of K decreases from 100 to 87. In each iteration, the value of K is printed. The value of K is printed $\left\lceil \frac{87-100}{-1} \right\rceil + 1 = 14$ times. Since K is the index of the loop, the value printed here is the value of the index, which varies in each iteration. Consider the following explicit **DO** loop version of the implied loop :

```
      DO 60 K = 100, 87 , -1
         PRINT*, K
60    CONTINUE
```

Output:

```
100
 99
 98
...
...
...
 87
```

The two loops are equivalent except in terms of the shape of the output. In the implied loop version, the output will be printed on one line. In the explicit **DO** loop version, the output will be printed as one value on each line.

Example 2: *Printing more than one value in each iteration of an implied loop: The following segment prints a percentage sign followed by a + sign three times :*

```
      PRINT*, ('%' , '+' , M = 1 , 3)
```

This produces the following output:

```
%+%+%+
```

Notice that the parenthesis encloses both the % and the + signs, which means they both have to be printed in every iteration the loop makes.

Example 3: Nested Implied Loops: *An implied loop may be nested either in another implied loop or in an explicit* ***DO*** *loop. There is no restriction on the number of levels of nesting. The following segment shows nested implied loops.*

```
      PRINT*, ((K, K = 1 , 5 , 2), L = 1 , 2)
```

Nested implied loops work in a similar manner as the nested **DO** loops. One very important point to note here is the double parenthesis before the K in the implied version. It means that the inner loop with index variable K is enclosed within the outer one with index variable L. The L loop is executed $\left\lceil \frac{2-1}{1} \right\rceil + 1 = 2$ times. The K loop forces the value of K to be printed $\left\lceil \frac{5-1}{2} \right\rceil + 1 = 3$ iterations. However, since the K loop is nested inside the L loop, the K loop is executed 3 times in each iteration of the L loop. Thus, K is printed 6 times. Therefore, the output of the implied version is:

```
1    3    5    1    3    5
```

5.7 Repetition Constructs in Subprograms

Subprograms in FORTRAN are considered separate programs during compilation. Therefore, repetition constructs in subprograms are given the same treatment as in programs. The following is an example that shows how repetition is used in subprograms.

Example: *Count of Integers in some Range that are Divisible by a given Value: Write a function subprogram that receives three integers as input. The* ***first*** *and* ***second*** *input integers make the range of values in which the function will conduct the search. The function searches for the integers in that range that are divisible by the* ***third*** *input integer. The function returns the count of such integers to the main program. The main program reads five lines of input. Each line consists of three integers. After each read, the main program calls the function, passes the three integers to it and receives the output from it and prints that output with a proper message :*

Solution:

```
      INTEGER K, L, M, COUNT, J, N
      DO 10 J = 1 , 5
         READ*, K, L, M
         N = COUNT(K , L , M)
         PRINT*, 'COUNT OF INTEGERS BETWEEN',K,'AND', L
         PRINT*, 'THAT ARE DIVISIBLE BY', M, 'IS', N
         PRINT*
10    CONTINUE
      END
      INTEGER FUNCTION COUNT(K , L , M)
      INTEGER K, L, M, INCR, NUM, J
      INCR = 1
      NUM = 0
      IF (L .LT. K) INCR = -1
      DO 10 J = K, L, INCR
         IF (MOD(J , M) .EQ. 0) NUM = NUM + 1
10    CONTINUE
      COUNT = NUM
      RETURN
      END
```

If we use the following input:

```
2   34  2
-15 -30  5
70  32   7
0   20   4
-10 10  10
```

The typical output would be as follows:

```
COUNT OF INTEGERS BETWEEN 2 AND 34
THAT ARE DIVISIBLE BY 2 IS 12

COUNT OF INTEGERS BETWEEN -15 AND -30
THAT ARE DIVISIBLE BY 5 IS 4

COUNT OF INTEGERS BETWEEN 70 AND 32
THAT ARE DIVISIBLE BY 7 IS 6

COUNT OF INTEGERS BETWEEN 0 AND 20
THAT ARE DIVISIBLE BY 4 IS 6

COUNT OF INTEGERS BETWEEN -10 AND 10
THAT ARE DIVISIBLE BY 10 IS 3
```

Remember what we said about the subprogram being a separate entity from the main program invoking it. Accordingly, note the following in the above example:

- It is allowed to use the same statement number in the main program and subprograms of the same file. Notice the statement number **10** in both the main program and the function subprogram
- It is also allowed to use the same variable name as index of **DO** loops in the main program and the subprogram. Notice the variable **J** in the above

5.8 Exercises

1. What will be printed by the following programs?

```
1.     LOGICAL FUNCTION PRIME(K)
       INTEGER N, K
       PRIME = .TRUE.
       DO 10 N = 2, K / 2
          IF (MOD(K , N) .EQ. 0) THEN
             PRIME = .FALSE.
          ENDIF
10     CONTINUE
       RETURN
       END
       LOGICAL PRIME
       PRINT*, PRIME(5), PRIME(8)
       END
```

```
2.     INTEGER FUNCTION FACT(K)
       INTEGER K,L
       FACT = 1
       DO 10 L = 2 , K
          FACT = FACT * L
10     CONTINUE
       RETURN
       END
       INTEGER FUNCTION COMB(N , M)
       INTEGER FACT
       IF (N .GT.M) THEN
          COMB = FACT(N) / (FACT(M) * FACT(N-M))
       ELSE
          COMB = 0
       ENDIF
       RETURN
       END
       INTEGER COMB
       PRINT*, COMB(4 , 2)
       END
```

```
3.     INTEGER K, M, N
       N = 0
       DO 10 K = -5 , 5
          N = N + 2
          DO 20 M = 3 , 1
             N = N + 3
20        CONTINUE
          N = N + 1
10     CONTINUE
       PRINT*, N
       END
```

```
4.    INTEGER ITOT, N
      READ*, N
      ITOT = 1
10    IF (N .NE. 0) THEN
         ITOT = ITOT * N
         READ*, N
         GOTO 10
      ENDIF
      READ*, N
20    IF (N .NE. 0) THEN
         ITOT = ITOT * N
         READ*, N
         GOTO 20
      ENDIF
      PRINT*,ITOT
      END
```

Assume the input is

```
2
0
3
0
4
```

```
5.    INTEGER FUNCTION CALC(A,B)
      INTEGER A,B,R, K
      R = 1
      DO 10 K=1,B
         R = R*A
10    CONTINUE
      CALC = R
      RETURN
      END
      INTEGER CALC
      READ*,M,N
      PRINT*,CALC(M,N)
      END
```

Assume the input is

```
2   5
```

```
6.    INTEGER KK, J, K
         KK = 0
2     IF ( KK.LE.0) THEN
         READ*, J , K
         KK = J - K
         GOTO 2
      ENDIF
      PRINT*,KK,J,K
      END
```

Assume the input is

```
2    3
-1    2
3    3
```

```
4    -3
2    5
4    3
```

```
7.     INTEGER K, J
       K = 2
25     IF ( K.GT.0 ) THEN
           DO 15 J = K, 3, 2
               PRINT*, K, J
15         CONTINUE
           K = K - 1
           GOTO 25
       ENDIF
       END
```

```
8.     INTEGER N, C
       LOGICAL FLAG
       READ*, N
       FLAG = .TRUE.
       C = N ** 2
22     IF ( FLAG ) THEN
           C = ( C + N ) / 2
           FLAG = C.NE.N
           PRINT*, C
           GOTO 22
       ENDIF
       END
```

Assume the input is

```
4
```

```
9.     INTEGER N, K
       READ*, N
       K = SQRT(REAL(N))
33     IF ( K*K .LT. N ) THEN
           K = K + 1
           GOTO 33
       ENDIF
       PRINT*, K*K
       END
```

Assume the input is

```
 6
```

```
10.    INTEGER J, K
       DO 10 K = 1,2
           PRINT*, K
           DO 10 J = 1,3
10             PRINT*,K,J
       END
```

```
11.    INTEGER X, K, M
       M = 4
       DO 100 K = M ,M+2
           X = M + 2
           IF ( K.LT.6 ) THEN
               PRINT*,'HELLO'
           ENDIF
100    CONTINUE
       END
```

```
12.    INTEGER SUM, K, J, M
       SUM = 0
       DO 1 K = 1,5,2
           DO 2 J = 7,-2,-3
               DO 3 M = 1980,1989,2
                   SUM = SUM + 1
3              CONTINUE
2          CONTINUE
1      CONTINUE
       PRINT*,SUM
       END
```

```
13.    LOGICAL T, F
       INTEGER BACK, FUTURE, K
       BACK = 1
       FUTURE = 100
       T = .TRUE.
       F = .FALSE.
       DO 99 K = BACK,FUTURE,5
           T = ( T.AND..NOT.T ) .OR. ( F.OR..NOT.F )
           F = .NOT.T
           FUTURE = FUTURE*BACK*(-1)
99     CONTINUE
       IF (T) PRINT*, 'DONE'
       IF (F) PRINT*, 'UNDONE'
       END
```

2. Find the number of iterations of the WHILE-LOOPS in each of the following programs:

```
1.     INTEGER K, M, J
       K = 80
       M = 5
       J = M-M/K*K
10     IF ( J.NE.0 ) THEN
           PRINT*, J
           J = M-M/K*K
           M = M + 1
           GOTO 10
       ENDIF
       END
```

```
2.     REAL W
       INTEGER L
       W = 2.0
       L = 5 * W
100    IF ( L/W.EQ.((L/4.0)*W) ) THEN
           PRINT*, L
           L = L + 10
           GOTO 100
       ENDIF
       END
```

3. Which of the following program segments causes an infinite loop?

```
(I)    J = 0
25     IF ( J.LT.5 ) THEN
           J = J + 1
           GOTO 25
       ENDIF
       PRINT*, J
```

```
II.    J = 0
25     IF ( J.LT.5 ) THEN
           J = J + 1
       ENDIF
       GOTO 25
       PRINT*, J
```

```
III.   X = 2.0
5      X = X + 1
       IF ( X.GT.4 ) X = X + 1
       GOTO 5
       PRINT*, X
```

```
IV.    M = 2
       K = 1
10     IF ( K.LE. M ) THEN
20         M = M + 1
           K = K + 2
           GOTO 20
       ENDIF
       GOTO 10
```

```
V.     X = 1
4      IF ( X.GE.1 ) GOTO 5
5      IF ( X.LE.1 ) GOTO 4
```

```
VI.    J = 1
33     IF ( J.GT.5 ) THEN
           GOTO 22
       ENDIF
       PRINT*, J
       J = J + 1
       GOTO 33
22     STOP
```

4. Convert the following WHILE loops to **DO** loops :

```
I.     ID = N
10     IF ( ID.LE.891234 ) THEN
           PRINT*, ID
           ID = ID + 10
           GOTO 10
       ENDIF
```

```
II.    L = 1
       SUM =0
3      IF (L.LE.15) THEN
           J = -L
2          IF (J.LE.0) THEN
               SUM =SUM+J
               J = J + 1
               GOTO 2
           ENDIF
           L = L+3
           GOTO 3
       ENDIF
       PRINT*,SUM
```

5. What will be printed by the following program :

```
      INTEGER ISUM, K, N
      ISUM = 0
      READ*, N
      DO 6 K = 1,N
         ISUM = ISUM +(-1)**(K-1)
6     CONTINUE
      PRINT*, ISUM
      END
```

If the input is:

a.

```
9
```

b.

```
8
```

c.

```
51
```

d.

```
98
```

6. The following program segments may or may not have errors. Identify the errors (if any).

```
1.    INTEGER K, J
      DO 6 K = 1,4
         DO 7 J = K-1,K
              PRINT*, K
6        CONTINUE
7     CONTINUE
      END
```

```
2.    INTEGER K, J
      K = 10
      J = 20
1     IF ( J.GT. K ) THEN
         K = K/2
         GOTO 1
      ENDIF
      END
```

7. Write a FORTRAN 77 program to calculate the following summation:

$$\sum_{k=1}^{200}\left((-1)^k \frac{5k}{k+1}\right)$$

8. Write a program that reads the values of two integers M and then prints all the odd numbers between the two integers.(Note: M may be less than or equal to N or vice-versa).

9. Write a program that prints all the numbers between two integers M and N which are divisible by an integer K. The program reads the values of M, N and K.

10. Write a program that prints all the perfect squares between two integers M and N. Your program should read the values of M and N. (Note: A perfect square is a square of an integer, example 25 = 5 × 5)

11. Using nested WHILE loops, print the multiplication table of integers from 1 to 10. Each multiplication table goes from 1 to 20. Your output should be in the form :

```
1 * 1 = 1
1 * 2 = 2
:
1 * 20 = 20
:
10 * 1 = 10
10 * 2 = 20
:
10 * 20 = 200
```

12. Rewrite the program in the previous question using nested **DO** loops.

13. Complete the **PRINT** statement in the following program to produce the indicated output.

```
      DO 1 K = 1,5
         PRINT*,
1     CONTINUE
      END
```

OUTPUT:

```
=****
*=***
**=**
***=*
****=
```

14. Complete the following program in order to get the required output.

```
      DO 10 K = 10,   (1)      ,   (2)
         PRINT*,(  (3)   , L =   (4)   , K )
10    CONTINUE
      END
```

The required output is :

```
5       6   7   8   9   10
5       6   7   8   9
5       6   7   8
5       6   7
5       6
5
```

5.9 Solutions to Exercises

Ans 1.

```
       T      F
       12
       33
       6
       25
       7      4      -3
       1 0    50
       10
   7
   5
   4
```

```
    9
    1
1 1
1 2
1 3
2
2 1
2 2
2 3
    HELLO
HELLO
60
DONE
```

Ans 2.

1. 76
2. INFINITE LOOP

Ans 3.

II , III , IV , V

Ans 4.

I)

```
      DO 10 ID = N , 891234 , 10
         PRINT*, ID
10    CONTINUE
```

II)

```
      SUM = 0
      DO 3 L = 1 , 15 , 3
         DO 2 J = -L , 0 , 1
            SUM = SUM + J
2        CONTINUE
3     CONTINUE
```

Ans 5.

A) 1 B) 0 C) 1 D) 0

Ans 6

1) IMPROPER NESTING OF **DO** LOOPS
2) INFINITE LOOP

Ans 7.

```
      REAL SUM
      INTEGER K
      SUM = 0
      DO 10 K = 1 , 200
         SUM = SUM + (-1) ** K * (REAL(5*K) / ( K+1))
10    CONTINUE
      PRINT*, SUM
      END
```

Ans 8.

```
      INTEGER M , N , TEMP
      READ*, M , N
      IF( M .LT. N ) THEN
         TEMP = N
         N = M
         M = TEMP
      ENDIF
      DO 5 L = M , N
         IF( L/2 * 2 .NE. L ) PRINT*,L
5     CONTINUE
      END
```

Ans 9.

```
      INTEGER M , N , K , TEMP
      READ*, M , N , K
      IF( M .LT. N ) THEN
         TEMP = N
         N = M
         M = TEMP
      ENDIF
      DO 5 L = M , N
         IF( L/K * K .EQ. L ) PRINT*,L
5     CONTINUE
      END
```

Ans 10.

```
      INTEGER M , N , TEMP
      READ*, M , N
      IF( M .LT. N ) THEN
         TEMP = N
         N   = M
         M   = TEMP
      ENDIF
      DO 5 L = M , N
         IF( INT(SQRT(REAL(L)) ** 2 .EQ. L )) PRINT*,L
5     CONTINUE
      END
```

Ans 11.

```
      INTEGER I, J
      I = 1
10    IF(I .LE. 10 ) THEN
          J = 1
5         IF( J .LE. 20 ) THEN
              PRINT*, I, ' * ', J, ' = ', I*J
              J = J + 1
              GO TO 5
          ENDIF
          I = I + 1
          GO TO 10
      ENDIF
      END
```

Ans 12.

```
      INTEGER I, J
      DO 10 I = 1 , 10
          DO 10 J = 1 , 20
              PRINT*, I, ' * ', J, ' = ', I*J
10    CONTINUE
      END
```

Ans 13.

```
PRINT*, ('*', J = 1, K-1), '=' , ('*', M = 1 , 5-K)
```

Ans 14.

1) 5 2) -1 3) L 4) 5