Counting Loop: `DO-END DO`

Syntax

Form 1

```
DO var = initial-value, final-value, step-size
   statements
END DO
```

Form 2

If `step-size` is 1, use

```
DO var = initial-value, final-value
   statements
END DO
```

- `var` is a variable of type `INTEGER`.
- `initial-value`, `final-value` and `step-size` are `INTEGER` expressions.
- For each value of the `var`, the body of the `DO` loop (*i.e.*, the `statements`) is executed once.
- The values for the `var` are `initial-value`, `initial-value + step-size`, `initial-value + 2*step-size` and so on until it is larger than the `final-value`.

Syntax Examples

1.

```
INTEGER :: Counter, Init, Final, Step

READ(*,*)  Init, Final, Step
DO Counter = Init, Final, Step
   .....
END DO
```

2.

```
INTEGER :: i, Lower, Upper

Lower = ....
Upper = ....
DO i = Upper - Lower, Upper + Lower
   .....
END DO
```

Semantics

- Before the `DO`-loop starts, the values of `initial-value`, `final-value` and `step-size` are computed **exactly ONCE**.
- The value of `step-size` cannot be zero.
- If the value of `step-size` is positive (counting up):
  1. `var` receives the value of `initial-value`;
  2. If `var` $\leq$ `final-value`, execute the statements in the body. Then, add the value of `step-size` to `var`. Go back to compare `var` and `final-value`.
  3. If `var` $>$ `final-value`, the `DO` loop completes.
- If the value of `step-size` is negative (counting down):
  1. `var` receives the value of `initial-value`;
  2. If `var` $\geq$ `final-value`, execute the statements in the body. Then, add the value of `step-size` to `var`. Go back to compare `var` and `final-value`.
  3. If `var` $<$ `final-value`, the `DO` loop completes.
- **DO NOT** change the value of `var` and any variable involved in the expressions `initial-value`, `final-value` and `step`. Or, you might be in **BIG** trouble!!!

Good Examples

1. The following `WRITE` produces -3, 9, -27 on the first row, -1, 1, -1 on the second, 1, 1, 1 on the third and 3, 9, 7 on the fourth.

```
INTEGER  :: Count

DO Count = -3, 4, 2
   WRITE(*,*)  Count, Count*Count, Count*Count*Count
END DO
```

2. The following `WRITE` displays 3, 4, and 5 from variable `Iteration`.

```
INTEGER, PARAMETER :: Init = 3, Final = 5
INTEGER            :: Iteration

DO Iteration = Init, Final
   WRITE(*,*)  'Iteration ', Iteration
END DO
```

3. If `a`, `b` and `c` receive 2, 7 and 5, then `MAX(a,b,c)` and `MIN(a,b,c)` are 7 and 2, respectively. Thus, variable `List` starts with 7 and counts down with values 7, 5 and 3.

```
INTEGER :: a, b, c
INTEGER :: List

READ(*,*)  a, b, c
DO List = MAX(a, b, c), MIN(a, b, c), -2
   WRITE(*,*)  List
END DO
```

More Examples

1. Suppose the value of `Number` is 10. The following code reads 10 integer values and add them together to `Sum`.

   ```
   INTEGER :: Count, Number, Sum, Input

   Sum = 0
   DO Count = 1, Number
      READ(*,*)  Input
      Sum = Sum + Input
   END DO
   ```

2. If you know adding numbers, you should know how to compute their average:

   ```
   INTEGER :: Count, Number, Sum, Input
   REAL    :: Average

   Sum = 0
   DO Count = 1, Number
      READ(*,*)  Input
      Sum = Sum + Input
   END DO
   Average = REAL(Sum) / Number
   ```

3. And, computing the product of numbers is very similar. The following computes the factorial of n, $n!$:

   ```
   INTEGER :: Factorial, N, I

   Factorial = 1
   DO I = 1, N
      Factorial = Factorial * I
   END DO
   ```

Something You Should Be Very Careful

1. `step-size` cannot be zero

```
INTEGER :: count

DO count = -3, 4, 0
   ...
END DO
```

2. Do not change the value of `var`

```
INTEGER :: a, b, c

DO a = b, c, 3
   READ(*,*)  a          ! the value of a is changed
   a = b-c               ! the value of a is changed
END DO
```

3. Do not change the value of any variable involved in the `initial-value`, `final-value` and `step-size`:

```
INTEGER :: a, b, c, d, e

DO a = b+c, c*d, (b+c)/e
   READ(*,*) b           ! initial-value is changed
   d = 5                 ! final-value is changed
   e = -3                ! step-size is changed
END DO
```

4. When you have a count-down loop, make sure the `step-size` is negative. The loop body of the following loop will **NOT** be executed. Why?

```
INTEGER :: i

DO i = 10, -10
   .....
END DO
```

5. While you can use `REAL` type for `control-var`, `initial-value`, `final-value` and `step-size`, it would be better not to use this feature at all, since it may be dropped in future FORTRAN standard. In the following, `x` successively receives -1.0, -0.75, -0.5, -0.25, 0.0, 0.25, 0.5, 0.75 and 1.0.

```
REAL :: x

DO x = -1.0, 1.0, 0.25
   .....
END DO
```

Programming Example 1

Read in a set of integers and count the number of positive, negative and zero input items.

```
PROGRAM  Counting
   IMPLICIT  NONE
   INTEGER :: Positive, Negative, PosSum, NegSum
   INTEGER :: TotalNumber, Count, Data

   Positive = 0
   Negative = 0
   PosSum   = 0
   NegSum   = 0
   READ(*,*)  TotalNumber
   DO Count = 1, TotalNumber
      READ(*,*)  Data
      WRITE(*,*) 'Input data ', Count, ': ', Data
      IF (Data > 0) THEN
         Positive = Positive + 1
         PosSum   = PosSum + Data
      ELSE IF (Data < 0) THEN
         Negative = Negative + 1
         NegSum   = NegSum + Data
      END IF
   END DO

   WRITE(*,*)  'Counting Report:'
   WRITE(*,*)  '   Positive items = ', Positive, ' Sum = ', PosSum
   WRITE(*,*)  '   Negative items = ', Negative, ' Sum = ', NegSum
   WRITE(*,*)  '   Zero items     = ', TotalNumber-Positive-Negative
   WRITE(*,*)
   WRITE(*,*)  'The total of all input is ', Positive + Negative

END PROGRAM  Counting
```

Programming Example 2

Compute the arithmetic, geometric and harmonic means and ignore all non-positive input items.

```
PROGRAM   ComputingMeans
   IMPLICIT  NONE
   REAL    :: X, Sum, Product, InverseSum
   REAL    :: Arithmetic, Geometric, Harmonic
   INTEGER :: Count, TotalNumber, TotalValid

   Sum        = 0.0
   Product    = 1.0
   InverseSum = 0.0
   TotalValid = 0
   READ(*,*)  TotalNumber
   DO Count = 1, TotalNumber
      READ(*,*)  X
      IF (X <= 0.0) THEN
         WRITE(*,*) 'Input <= 0.  Ignored'
      ELSE
         TotalValid = TotalValid + 1
         Sum        = Sum + X
         Product    = Product * X
         InverseSum = InverseSum + 1.0/X
      END IF
   END DO
   IF (TotalValid > 0) THEN
      Arithmetic = Sum / TotalValid
      Geometric  = Product**(1.0/TotalValid)
      Harmonic   = TotalValid / InverseSum
      WRITE(*,*)  'No. of valid items --> ', TotalValid
      WRITE(*,*)  Arithmetic, Geometric, Harmonic
   ELSE
      WRITE(*,*)  'ERROR: none of the input is positive'
   END IF
END PROGRAM  ComputingMeans
```

Programming Example 3

Compute the factorial of $n \geq 0$, $n!$, with a "bullet-proof" program so that your program could reject all negative input.

```
PROGRAM  Factorial
   IMPLICIT  NONE

   INTEGER :: N, i, Answer

   WRITE(*,*)  'This program computes the factorial of'
   WRITE(*,*)  'a non-negative integer'
   WRITE(*,*)
   WRITE(*,*)  'What is N in N! --> '
   READ(*,*)   N
   WRITE(*,*)

   IF (N < 0) THEN
      WRITE(*,*)  'ERROR: N must be non-negative'
      WRITE(*,*)  'Your input N = ', N
   ELSE IF (N == 0) THEN
      WRITE(*,*)  '0! = 1'
   ELSE
      Answer = 1
      DO i = 1, N
         Answer = Answer * i
      END DO
      WRITE(*,*)  N, '! = ', Answer
   END IF

END PROGRAM  Factorial
```

General DO-Loop with EXIT

The most general form of the `DO` statement is the following:

```
DO
   statements
END DO
```

This will cause the `statements` to be executed over and over without any chance to stop. To bail out from a `DO` loop, use the `EXIT` statement:

```
DO
   statements-1
   IF (logical-expression)  EXIT
   statements-2
END DO

DO
   statements-1
   IF (logical-expression)  THEN
      statements
      EXIT
   END IF
   statements-2
END DO
```

The `EXIT` statement brings the control of execution to the statement following the `END DO` statement, thus bailing out of the `DO` loop.

Examples

1. The following example reads a number of integers and computes their sum until a negative number occurs.

```
INTEGER :: x, Sum

Sum = 0
DO
   READ(*,*)  x
   IF (x < 0)  EXIT
   Sum = Sum + x
END DO
```

2. The following example shows how to write a counting loop with `REAL` numbers. Variable `x` receives values -1.0, -0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75 and 1.0

```
REAL, PARAMETER :: Lower = -1.0
REAL, PARAMETER :: Upper = 1.0
REAL, PARAMETER :: Step = 0.25
REAL            :: x

x = Lower
DO
   IF (x > Upper)  EXIT
   WRITE(*,*)  x
   x = x + Step
END DO
```

3. The following example asks the user to type in a number in the range of 0 and 10 inclusive. If the input is not in this range, the user will be asked again.

```
INTEGER :: Input

DO
   WRITE(*,*)  'An integer >= 0 and <= 10: '
   READ(*,*)   Input
   IF (0 <= Input .AND. Input <= 10)  EXIT
   WRITE(*,*)  'Out of range.  Try again'
END DO
```

Two Common Mistakes

1. The `EXIT` condition is `.FALSE.` forever. This could be a result of forgetting to update an involved variable. Here are two examples:

```
INTEGER  :: i

i = 5
DO
   IF (i < -2)  EXIT     ! i < -2 is ALWAYS .FALSE.
   WRITE(*,*)  i
END DO
```

```
INTEGER :: i = 1, j = 5

DO
   IF (j < 0)  EXIT      ! j < 0 is ALWAYS .FALSE.
   WRITE(*,*)  i
   i = i + 1
END DO
```

2. Did you initialize the control variable?

```
INTEGER :: i

DO
   IF (i <= 3)  EXIT     ! who knows what the
   WRITE(*,*)  i         ! result of i <= 3 is
   i = i - 1
END DO
```

Programming Example 1

Read in a set of integers until a negative one is encountered and find the maximum and minimum.

```
PROGRAM  MinMax
   IMPLICIT  NONE

   INTEGER :: Minimum, Maximum
   INTEGER :: Count
   INTEGER :: Input e

   Count = 0
   DO
      READ(*,*) Input
      IF (Input < 0)  EXIT
      Count = Count + 1
      WRITE(*,*)  'Data item #', Count, ' = ', Input
      IF (Count == 1) THEN
         Maximum = Input
         Minimum = Input
      ELSE
         IF (Input > Maximum)  Maximum = Input
         IF (Input < Minimum)  Minimum = Input
      END IF
   END DO

   WRITE(*,*)
   IF (Count > 0) THEN
      WRITE(*,*)  'Found ', Count, ' data items'
      WRITE(*,*)  '  Maximum = ', Maximum
      WRITE(*,*)  '  Minimum = ', Minimum
   ELSE
      WRITE(*,*)  'No data item found.'
   END IF

END PROGRAM  MinMax
```

Programming Example 2

Given a positive number b, its square root can be computed *iteratively* with the following formula:

$$\text{New } x = \frac{1}{2}\left(x + \frac{b}{x}\right)$$

where x starts with b. For the next iteration, the New x becomes x. This process continues until the absolute difference between x and New x is smaller than a given tolerance value.

```
PROGRAM  SquareRoot
   IMPLICIT  NONE

   REAL    :: Input, X, NewX, Tolerance
   INTEGER :: Count

   READ(*,*)  Input, Tolerance

   Count = 0
   X     = Input
   DO
      Count = Count + 1
      NewX  = 0.5*(X + Input/X)
      IF (ABS(X - NewX) < Tolerance)  EXIT
      X = NewX
   END DO

   WRITE(*,*)  'After ', Count, ' iterations:'
   WRITE(*,*)  '  The estimated square root is ', NewX
   WRITE(*,*)  '  The square root from SQRT() is ', SQRT(Input)
   WRITE(*,*)  '  Absolute error = ', ABS(SQRT(Input) - NewX)

END PROGRAM  SquareRoot
```

Programming Example 3

The exponential function `exp(x)` is usually defined to be the sum of the following infinite series:

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots$$

Use this series to compute `exp(x)` until the absolute value of a term is less than a tolerance value, say 0.00001

```
PROGRAM  Exponential
   IMPLICIT  NONE

   INTEGER          :: Count
   REAL             :: Term
   REAL             :: Sum
   REAL             :: X
   REAL, PARAMETER :: Tolerance = 0.00001

   READ(*,*)  X
   Count = 1
   Sum   = 1.0
   Term  = X
   DO
      IF (ABS(Term) < Tolerance)  EXIT
      Sum   = Sum + Term
      Count = Count + 1
      Term  = Term * (X / Count)
   END DO

   WRITE(*,*)  'After ', Count, ' iterations:'
   WRITE(*,*)  '  Exp(', X, ') = ', Sum
   WRITE(*,*)  '  From EXP()   = ', EXP(X)
   WRITE(*,*)  '  Abs(Error)   = ', ABS(Sum - EXP(X))

END PROGRAM  Exponential
```

Programming Example 4

The *Greatest Common Divisor*, GCD for short, of two positive integers can be computed with Euclid's division algorithm. Let the given numbers be a and b, $a \geq b$. Euclid's division algorithm has the following steps:

1. Compute the remainder c of dividing a by b.
2. If the remainder c is zero, b is the greatest common divisor.
3. If c is not zero, replace a with b and b with the remainder c. Go back to step (1).

```
PROGRAM  GreatestCommonDivisor
   IMPLICIT  NONE

   INTEGER   :: a, b, c

   WRITE(*,*) 'Two positive integers please --> '
   READ(*,*)  a, b
   IF (a < b) THEN         ! since a >= b must be true, they
      c = a                ! are swapped if a < b
      a = b
      b = c
   END IF

   DO                      ! now we have a <= b
      c = MOD(a, b)        !    compute c, the reminder
      IF (c == 0) EXIT     !    if c is zero, we are done.  GCD = b
      a = b                !    otherwise, b becomes a
      b = c                !    and c becomes b
   END DO                  !    go back

   WRITE(*,*) 'The GCD is ', b

END PROGRAM  GreatestCommonDivisor
```

Programming Example 5

An positive integer greater than or equal to 2 is a *prime* number if it is 2 or the only divisors of this integer are 1 and itself. Write a program that reads in an arbitrary integer and determines if it is a prime number.

```
PROGRAM  Prime
   IMPLICIT  NONE

   INTEGER  :: Number
   INTEGER  :: Divisor

   READ(*,*)  Number
   IF (Number < 2) THEN
      WRITE(*,*)  'Illegal input'
   ELSE IF (Number == 2) THEN
      WRITE(*,*)  Number, ' is a prime'
   ELSE IF (MOD(Number,2) == 0) THEN
      WRITE(*,*)  Number, ' is NOT a prime'
   ELSE
      Divisor = 3
      DO
         IF (Divisor*Divisor>Number .OR. MOD(Number,Divisor)==0) &
              EXIT
         Divisor = Divisor + 2
      END DO
      IF (Divisor*Divisor > Number) THEN
         WRITE(*,*)  Number, ' is a prime'
      ELSE
         WRITE(*,*)  Number, ' is NOT a prime'
      END IF
   END IF
END PROGRAM  Prime
```

Nested `DO-END DO`

Syntax

```
DO
   statements-1
   DO
      statements-2
   END DO
   statement-3
END DO
```

For each iteration, `statements-1` is executed, followed by the *inner* `DO`-loop, followed by `statements-3`.

Examples

1. The following example displays the value of 1*1, 1*2, 1*3, ..., 1*9, 2*1, 2*2, 2*3, ..., 2*9, 3*1, 3*2, ..., 3*9, ..., 9*1, 9*2, ..., 9*9.

```
INTEGER :: i, j

DO i = 1, 9
   DO j = 1, 9
     WRITE(*,*)  i*j
   END DO
END DO
```

2. The following example displays 4, 3, 5; 4, 8, 10; 12, 5, 13; 8, 15, 17; ..., and 40, 9, 41.

```
INTEGER :: u, v
INTEGER :: a, b, c

DO u = 2, 5
   DO v = 1, u-1
      a = 2*u*v
      b = u*u - v*v
      c = u*u + v*v
      WRITE(*,*)  a, b, c
   END DO
END DO
```

3. The following example computes 1, 1+2, 1+2+3, 1+2+3+4, ..., 1+2+3+...+9.

```
INTEGER :: i, j, Sum

DO i = 1, 10
   Sum = 0
   DO j = 1, i
      Sum = Sum + j
   END DO
   WRITE(*,*)  Sum
END DO
```

4. The following example computes the square roots of 0.1, 0.2, 0.3, ..., 0.9 with Newton's method.

```
REAL :: Start = 0.1, End = 1.0, Step = 0.1
REAL :: X, NewX, Value

Value = Start
DO
   IF (Value > End)  EXIT
   X = Value
   DO
      NewX = 0.5*(X + Value/X)
      IF (ABS(X - NewX) < 0.00001)  EXIT
      X = NewX
   END DO
   WRITE(*,*)  'The square root of ', Value, ' is ', NewX
   Value = Value + Step
END DO
```

Programming Example 1

There are four sessions of CS110 and CS201, each of which has a different number of students. Suppose all students take three exams. Someone has prepared a file that records the exam scores of all students. This file has a form as follows:

```
4
3
97.0  87.0  90.0
100.0 78.0  89.0
65.0  70.0  76.0
2
100.0 100.0 98.0
97.0  85.0  80.0
4
78.0  75.0  90.0
89.0  85.0  90.0
100.0 97.0  98.0
56.0  76.0  65.0
3
60.0  65.0  50.0
100.0 99.0  96.0
87.0  74.0  81.0
```

Write a program that reads in a file of this form and computes the following information: **(1)** the average of each student; **(2)** the class average of each exam; and **(3)** the grant average of the class.

```
PROGRAM  ClassAverage
   IMPLICIT  NONE

   INTEGER :: NoClass
   INTEGER :: NoStudent
   INTEGER :: Class, Student
   REAL    :: Score1, Score2, Score3, Average
   REAL    :: Average1, Average2, Average3, GrantAverage

   READ(*,*)  NoClass
   DO Class = 1, NoClass
      READ(*,*)  NoStudent
      WRITE(*,*)
      WRITE(*,*) 'Class ', Class, ' has ', NoStudent, ' students'
      WRITE(*,*)
      Average1 = 0.0
      Average2 = 0.0
      Average3 = 0.0
      DO Student = 1, NoStudent
         READ(*,*)  Score1, Score2, Score3
         Average1 = Average1 + Score1
         Average2 = Average2 + Score2
         Average3 = Average3 + Score3
         Average  = (Score1 + Score2 + Score3) / 3.0
         WRITE(*,*)  Student, Score1, Score2, Score3, Average
      END DO
      WRITE(*,*)  '----------------------'
      Average1     = Average1 / NoStudent
      Average2     = Average2 / NoStudent
      Average3     = Average3 / NoStudent
      GrantAverage = (Average1 + Average2 + Average3) / 3.0
      WRITE(*,*) 'Class Average: ', Average1, Average2, Average3
      WRITE(*,*) 'Grant Average: ', GrantAverage
   END DO

END PROGRAM  ClassAverage
```

Programming Example 2

The exponential function `exp(x)` is usually defined to be the sum of the following infinite series:

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^i}{i!} + \cdots$$

Write a program to read in an initial value, a final value and a step size, and computes `exp(x)`.

```
PROGRAM  Exponential
   IMPLICIT  NONE

   INTEGER         :: Count
   REAL            :: Term, Sum, X, ExpX, Begin, End, Step
   REAL, PARAMETER :: Tolerance = 0.00001

   WRITE(*,*)  'Initial, Final and Step please --> '
   READ(*,*)   Begin, End, Step
   X = Begin
   DO
      IF (X > End)  EXIT
      Count = 1
      Sum   = 1.0
      Term  = X
      ExpX  = EXP(X)
      DO
         IF (ABS(Term) < Tolerance)  EXIT
         Sum   = Sum + Term
         Count = Count + 1
         Term  = Term * (X / Count)
      END DO
      WRITE(*,*)  X, Sum, ExpX, ABS(Sum-ExpX), ABS((Sum-ExpX)/ExpX)
      X = X + Step
   END DO
END PROGRAM  Exponential
```

Programming Example 3

An *Armstrong number* of three digits is an integer such that the sum of the cubes of its digits is equal to the number itself. For example, 371 is an Armstrong number since 3**3 + 7**3 + 1**3 = 371. Write a program to find all Armstrong number in the range of 0 and 999.

```
PROGRAM  ArmstrongNumber
   IMPLICIT  NONE

   INTEGER :: a, b, c
   INTEGER :: abc, a3b3c3
   INTEGER :: Count

   Count = 0
   DO a = 0, 9
      DO b = 0, 9
         DO c = 0, 9
            abc    = a*100 + b*10 + c
            a3b3c3 = a**3 + b**3 + c**3
            IF (abc == a3b3c3) THEN
               Count = Count + 1
               WRITE(*,*)  'Armstrong number ', Count, &
                           ': ', abc
            END IF
         END DO
      END DO
   END DO

END PROGRAM  ArmstrongNumber
```

Programming Example 4

Write a program to read a value for n, make sure that n is greater than or equal to 2, and display all prime numbers in the range of 2 and n. In case n is less than 2, your program should keep asking the user to try again until a value that is greater than or equal to 2 is read.

Programming ideas:

1. 2 is a prime number
2. All even numbers are **not** primes
3. Only odd numbers are tested
4. For each odd number M, use 3, 5, 7, 9, 11, ..., $\sqrt{M}$ to test if they evenly divide M.
   (a) If none of these numbers can divide M, M is a prime
   (b) Otherwise, M is not a prime. Proceed to test $M+2$.

```
PROGRAM  Primes
   IMPLICIT  NONE

   INTEGER  :: Range, Number, Divisor, Count

   WRITE(*,*)  'What is the range ? '
   DO
      READ(*,*)  Range
      IF (Range >= 2)  EXIT
      WRITE(*,*)  'The range value must be >= 2.'
      WRITE(*,*)  'Please try again:'
   END DO

   Count = 1
   WRITE(*,*)
   WRITE(*,*)  'Prime number #', Count, ': ', 2
   DO Number = 3, Range, 2

      Divisor = 3
      DO
         IF (Divisor*Divisor>Number .OR. MOD(Number,Divisor)==0) &
              EXIT
         Divisor = Divisor + 2
      END DO

      IF (Divisor*Divisor > Number) THEN
         Count = Count + 1
         WRITE(*,*)  'Prime number #', Count, ': ', Number
      END IF
   END DO

   WRITE(*,*)
   WRITE(*,*)  'There are ', Count, &
               ' primes in the range of 2 and ', Range

END PROGRAM  Primes
```

Programming Example 5

Write a program to find all prime factors of a positive integer. For example, since we have

$$586390350 = 2 \times 3 \times 5^2 \times 7^2 \times 13 \times 17 \times 19^2$$

your program should report the following factors:

$$2, 3, 5, 5, 7, 7, 13, 17, 19, 19$$

Programming ideas:

1. Remove all factors of 2 first.
2. Use 3, 5, 7, 9, 11, 13, 15, ... to try if they are factors.
3. If k is a factor, remove it.
4. How to remove a factor $k = 3$ from $n = 135$?
   (a) Use k to divide n repeatedly and use the quotient to replace n.
   (b) Dividing 135 by 3 yields a quotient 45. The new n is 45.
   (c) Dividing 45 by 3 yields a quotient of 15. The new n is 15.
   (d) Dividing 15 by 3 yields a quotient 5. The new n is 15.
   (e) Since 5 cannot be divided by 3, we are done and three factors of 3 have been removed.

```
PROGRAM  Factorize
   IMPLICIT  NONE

   INTEGER  :: Input
   INTEGER  :: Divisor
   INTEGER  :: Count

   READ(*,*)   Input

   Count = 0
   DO
      IF (MOD(Input,2) /= 0 .OR. Input == 1)  EXIT
      Count = Count + 1
      WRITE(*,*)  'Factor # ', Count, ': ', 2
      Input = Input / 2
   END DO

   Divisor = 3
   DO
      IF (Divisor > Input) EXIT
      DO
         IF (MOD(Input,Divisor)/=0 .OR. Input==1) EXIT
         Count = Count + 1
         WRITE(*,*)  'Factor # ', Count, ': ', Divisor
         Input = Input / Divisor
      END DO
      Divisor = Divisor + 2
   END DO

END PROGRAM  Factorize
```

The `IOSTAT=` Option in `READ(*,*)`

```
INTEGER :: IOstatus

READ(*,*,IOSTAT=IOstatus) var1, ..., varn
```

- The variable following `IOSTAT=` must be of type `INTEGER`
- After executing `READ(*,*,IOSTAT=var)`, `var` receives a value:
  - If this value is zero, everything was fine.
  - If this value is negative, the end of file has reached. That is, no more data in a file.
  - If this value is positive, something was wrong in the input.
- To generate the end of file signal with your keyboard, use **Ctrl-D**.

Examples

1. After executing `READ(*,*,IOSTAT=Reason)`, one should test the value of `Reason` and find out the reason:

```
INTEGER :: Reason
INTEGER :: a, b, c

DO
   READ(*,*,IOSTAT=Reason)  a, b, c
   IF (Reason > 0)  THEN
      ... something wrong ...
   ELSE IF (Reason < 0) THEN
      ... end of file reached ...
   ELSE
      ... do normal stuff ...
   END IF
END DO
```

2. The following reads in integers and computes their sum in `sum`. If something is wrong or end of file is reached, exit the loop.

```
INTEGER :: io, x, sum

sum = 0
DO
   READ(*,*,IOSTAT=io)  x
   IF (io > 0) THEN
      WRITE(*,*) 'Check input.  Something was wrong'
      EXIT
   ELSE IF (io < 0) THEN
      WRITE(*,*)  'The total is ', sum
      EXIT
   ELSE
      sum = sum + x
   END IF
END DO
```

Programming Example

The arithmetic mean (*i.e.*, average), geometric mean and harmonic mean of a set of n numbers $x_1 1, x_2, ..., x_n$ is defined as follows:

$$\begin{aligned}
\text{Arithmetic Mean} &= \frac{1}{n}(x_1 + x_2 + \cdots + x_n) \\
\text{Geometric Mean} &= \sqrt[n]{x_1 \times \times x_2 \times \cdots \times x_n} \\
&= (x_1 \times x_2 \times \cdots \times x_n)^{1/n} \\
\text{HarmonicMean} &= \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n}}
\end{aligned}$$

Since computing geometric mean requires taking root, it is further required that all input data values must be positive. As a result, this program must be able to ignore non-positive items. However, this may cause **all** input items ignored. Therefore, before computing the means, this program should have one more check to see if there are valid items.

This program should be capable of reporting input error. For example, if the input contains a number `3.o` rather than `3.0`.

```
PROGRAM  ComputingMeans
   IMPLICIT  NONE
   REAL    :: X, Sum, Product, InverseSum
   REAL    :: Arithmetic, Geometric, Harmonic
   INTEGER :: Count, TotalValid, IO

   Sum        = 0.0
   Product    = 1.0
   InverseSum = 0.0
   TotalValid = 0
   Count      = 0

   DO
      READ(*,*,IOSTAT=IO)  X
      IF (IO < 0)  EXIT
      Count = Count + 1
      IF (IO > 0) THEN
         WRITE(*,*)  'ERROR: something wrong in input'
         WRITE(*,*)  'Try again please'
      ELSE
         WRITE(*,*) 'Input item ', Count, ' --> ', X
         IF (X <= 0.0) THEN
            WRITE(*,*) 'Input <= 0.  Ignored'
         ELSE
            TotalValid = TotalValid + 1
            Sum        = Sum + X
            Product    = Product * X
            InverseSum = InverseSum + 1.0/X
         END IF
      END IF
   END DO
```

```
   IF (TotalValid > 0) THEN
      Arithmetic = Sum / TotalValid
      Geometric  = Product**(1.0/TotalValid)
      Harmonic   = TotalValid / InverseSum
      WRITE(*,*)  '# of items read --> ', Count
      WRITE(*,*)  '# of valid items -> ', TotalValid
      WRITE(*,*)  'Arithmetic mean --> ', Arithmetic
      WRITE(*,*)  'Geometric mean  --> ', Geometric
      WRITE(*,*)  'Harmonic mean   --> ', Harmonic
   ELSE
      WRITE(*,*)  'ERROR: none of the input is positive'
   END IF
END PROGRAM  ComputingMeans
```