Fortran programming and numerical analysis: Paper Phy425

Programming - making the computer do a job for you.
A set of statements. Written in a particular language following some given grammatical rules.
Basic statements - **read, write, assignment, comment, stop** and **end** (for the time being)
A program uses **constants and variables** (remember your first lessons in algebra!)
Constants: Have a particular value: e.g., $2.0, -31.04, 4, 10^3, 0.001$ etc.
Variables: can assume an assigned value. Let a be a variable. Writing $a = 1.0$ implies the value 1.0 is assigned to the variable a. Writing $a = 5.5$ will result in a assuming a different value.

Assignment statements in general:

Variable = a constant or another variable or an expression involving constants and variables

Expression is a combination of arithmetical operations in general
Arithmetical operations:
• addition uses the plus symbol, +
• subtraction uses the dash or minus symbol, -
• multiplication uses the asterisk, *
• division uses the solidus slash, /
• exponentiation uses the double asterisk or **
Assignment statements: examples
a = 2
a= c3
a2 = 2*b -c/d
ax = 3 - 4*a_z

a = a+1
a = a-1

Yes, the last two statements are valid in Fortran! a = a+1 means the value of a is increased by 1. So if a was 3.5 previously, it is now 4.5
You can see that we have used **variable names** like a, a2, c3, a_z etc. A variable name always begins with an alphabet (fortran 77 does not distinguish between captical and small) and is followed by a numeral/alphabet or _ (underscore). Max length of a variable name is 6 in fortran 77 (more in other versions)
Precedence of operations:
• Negation (change of sign, not subtraction) is done first
• Exponentiation is done next.
• Multiplication and division are done next
• Addition and subtraction are done last

Example:
c = 2. + 3.1416 * 5. ** 2

First 5^2 is done, then the product with 2.1416 is made and finally 2 is added.
Parentheses can be used to rearrange precedence
c = (2. + (3.1416 * 5.)) ** 2
It is left to the student to find out the sequence of the operations here.
Only first bracket, (and), allowed in fortran.

Type of constants and variables:
Integer and Real
Integer constants are constant data elements that have no fractional parts or exponents. They always begin with a digit. Can be signed or unsigned. Examples: 1, -62, 1001 etc.
Real constants: have fractional parts or exponents e.g., 1.02, -65.452, 1.05×10^3 etc.
Integer variables : variables to which only values without fractional part can be assigned
real variables : variables which have fractional parts
Integer variables in fortran 77 : assume integer values only: in Fortran77 variable names beginning with i,j,k,l,m,n are by default integer variables , e.g., i, i3, index etc.
If one writes i = 2 or i= 2.5, i will assume the value 2 in both cases.
Real variables: Variable names beginning with a-h, o-z (by default)

a=2.5, b = 34573.424 etc.
Now we are in a position to write a short simple program involving simple arithmetical operations:

```
c       this is a program to add two numbers a and b and store it as c
        a =1.5
        b = 3.4
        c = a+ b
        write (*,*) 'values are', a,b,c
        stop
        end
```

c in the first column signifies a comment statement (does not execute anything), stop means all excecutions are stopped at that point and a program always ends with "end".
Stop is not a necessary statement but end is.
Stop can occur anywhere in the program but end can occur only as the last statement.
Write statement: Anything written within the single quotes ' and ' will be printed as it is.

Common unix commands
To edit a file (either new or old)

gedit filename.f

To compile (translate to machine language): gfortran filename.f
Or g90 filename.f

(There are other compilers also.)

Complilation successful: generates an **excutable file** a.out

To run the excutable file: ./a.out

If the compilation is unsuccessful, errors will be pointed out. Program should be re-edited and corrected AND compiled to get a.out.

Library functions

Some trigonometric or other functions are available in the fortran library
Examples : sin(x), cos(x) etc.
A list:

Function	Operation	Argument	Output
$\sin(x)$ $\cos(x)$ $\tan(x)$	sine cosine tangent of angle x	real in radian	real
$\exp(x)$	e^x	real	real
$\sinh(x)$ etc	hyperbolic sine of x real	real	
$a\sin(x)$ etc	$sin^{-1}(x)$ etc.	real	real
$\log(x)$	$log_e(x)$	real	real
$\log 10(x)$	$log_10(x)$	real	real
float(i)	convert to real	integer	real
mod(m,n)	remainder after m is divided by n	integer	integer
int(x) nint(x)	integer part of x nearest integer of x	real real	integer integer
sqrt(x)	$\sqrt{x}$	real	real
iabs(i) abs(x)	$\|i\|$ $\|x\|$	integer real	integer real

Read statements

So far you have just assigned values to the variables inside the programme (e.g., by writing $b = 2.4$ etc.) . If you want to change the values you have to edit the program, recompile it and run. To avoid this use a READ statement so that values can be assigned externally.

Example

```
c       this is a program to add two numbers a and b and store it as c
        read (*,*) a, b
c        a =1.5
c        b = 3.4
        c = a+ b
        write (*,*) 'values are', a,b,c
        stop
        end
```

How to assign the values? Type from the keyboard.

Mode of arithmetical operations

Integer mode, real mode and mixed mode

Integer mode: the operations are among integer variables and/or constants: result is integer
Examples: 2+10, 3*8, 50/9, i*j, 2*i1, i+j/i etc.
50/9 will simply result in 5 as the fractional part will be truncated.
Similarly if j = 15 and i=4 then j/i = 3 and NOT 3.75
One can write
k= j/i
or,
a= j/i
In case I, the value 3 is assigned to k
In case II, value 3 is assigned to a but if I print a it will come out as 3.0
So the mode of operation on the rhs is important.

Real mode: operations with reals: result is real
e.g., 2.466*153.75 + a*b1 - c*d
Suppose I write
a= 2.2*3.7
The value assigned to a will be 8.14
If I write
k = 2.2*3.7
The value of k will be just 8

Mixed mode: operations with both integer and real variables and constants. Result is real.
e.g., 2*3.2 = 6.4 etc.

Real and Integer declaration statements
Although by default variable names beginning with a-h, o-z are real and variable names beginning with i-n are integers, one can make variables real or integer by choice by using a declaration statement on top of the program.
e.g.,

```
Real i
j = 5
i = 3.4
```

Now if you have an operation i/j, it will be done in mixed mode and a nonzero value will result.
Similarly you can declare

```
integer x
x = 5
i = 3
```

Result of x/i will be simply 1.

Simple exercises:
Are these valid fortran variable names?
(a) xyigh (b) a+b (c) q@123 (d) t567 (e) 34xw (f) x/y (g) a_xar (h) resistance

Are these statements valid fortran statements? Correct them if they are not.
(a) x = x/y/x
(b) a+b = c
(c) dminus-1 = d - 1.0
(d) dminus_1 = d - 1.0
(e) a = a+ a + a
(f) c2 = 2x + y
(g) a= ((3*y -1) -(2-y)
(h) = { (a-b) + c}

Transferring of control
Unconditional go to statement.
Need to assign label to the statements
If you want to transfer the control, simply go to that statement.
Example:

```
      a= 1
      b=3
      go to 47
      b = 6
 47 write (*,*) a,b
```

Put labels in the first 1-5 columns.

Caution: go to statement not much encouraged nowadays!

A fortran statement is written between the 7-72 columns. What is the sixth column for? If the statement exceeds 72 columns continue on next line with a continuity index in the sixth column.

Conditional statement: IF (logical)

An important part of any programming language are the conditional statements. The most common such statement in Fortran is the if statement, which actually has several forms. The simplest one is the logical if statement:

Consider the problem of finding the roots of the equation

$$ax^2 + bx + c = 0.$$

If $b^2 - 4ac < 0$, you cannot find the roots and you must take this condition under consideration.
First way: simply stop if $b^2 - 4ac < 0$.
In fortran this will be written as

```
      if (b*b - 4*a*c.lt.0) stop
```

In general, a if statement is written as :

if (logical expression) executable statement

Logical expressions:
Comaparisons:

a.eq.b
a.ne.b
a.gt.b
a.lt.b
a.le.b
a.ge.b

Logical expression ≡ True or False (logical datatype)

If the logical expression is True, the statement following it will be executed,if it is False, the next statement will be executed.
Example:

```
read (*,*) a, b, c
if (b**2 - 4*a*c.lt.0) stop
root1 = ......
root2 = .....
```

etc.
Executable statements can be of various forms, e.g., stop, write or print, assignment statements, go to statements etc.
Examples:
a=4.5
b = 3.5
if (a.eq.b) write (*, *) 'a = b'

if (a.ne.b) go to 20

if (a.gt.b) x = x + 2

if (a.ge.b) print *, 'a is greater than OR equal to b'

If more than one statement should be executed inside the if, then the following syntax should be used:

```
if (logical expression) then
statements
endif
```

Sometimes when the if is satisfied a number of statements have to be executed, and, if not, then also a number of statements have to be executed. Then the syntax should be

```
if (logical expression) then
 statements
 else
   statements
  endif
```

If - then statement should always end with endif.

<u>If within if</u>:
Syntax:

```
if (logical expression 1) then
 statements
                if (logical expression 2) then
                 statements
                 endif
 else
   statements
  endif
```

The inner if has to be closed first. This is called a blocked structure (can contain even more ifs).

It is not possible to enter a if block from outside but it is possible to exit from a if block.
Examples:

```
          go to 12
          if (logical expression) then
         statements
12        statement
         endif
```

The above is NOT allowed.

But

```
          if (logical expression 1) then
                if (logical expression 2) go to 12
            statements
          endif
           statements
12         ...
```

is allowed.

Comparing two real constants or variable

Two real variables can never be checked for equalily. This is because their values might get altered due to rounding off etc. in the computer. So to check whether two variables, say, A and B are equal, check whether the **absolute value of their difference is less than** ϵ where ϵ has a small value (e.g., 10^{-8}).

Simple exercises:

1. Write a Fortran 77 code segment that assigns the real variable t the following value (assume x and y have been defined previously):

0, if x or y is zero
y, if x is negative
x+y, if x and y are both positive
x−y, if x is positive and y negative

2. Assume m and n have been defined previously. Write a Fortran 77 code segment that increases the real variable t by 0.5 if
(a) n is a factor of m
(b) m and n are both even
(c) m is odd but n is even

3. Write a Fortran 77 code segment for the following logical statements:
(a) If i,j,k are all equal, assign y to x
(b) If any one of i,j and k is equal to zero, print a statement that at least one of them is zero.

Some of the conditions have to done in two steps. Later we will learn to do things in a single step.

Other logical operations:

Logical AND A logical AND statement evaluates two expressions, and if both expressions are true, the logical AND statement is true as well.

Suppose x is to be incremented by 1 if both m and n (two preassigned integer variables) are both even numbers.

if (mod(n,2).eq.0.AND. mod(m.2).eq.0) x = x+1

Logical OR A logical OR statement evaluates two expressions. If either one is true, the expression is true.

Suppose y is to be added to x if either of m and n is even.

if (mod(n,2).eq.0.OR.mod(m.2).eq.0) x = x+y

Logical NOT A logical NOT statement evaluates true if the expression being tested is false. Again, if the expression being tested is false, the value of the test is TRUE.

If (.NOT.x.eq.5) .....

(same as if (x.ne.5))

So you have now learnt three kinds of operations:

ARITHMETIC (*, ** etc)

RELATIONAL (le, ge, gt, lt, eq, also written as $<=, >=, >, <, ==$)

LOGICAL .NOT., .AND., .OR., .EQV., .NEQV.

EQV gives .TRUE. when both logical variables being compared are both true or both false, NEQV does the opposite.

The most general form of the if statement

```
if (logical expression) then
statements
elseif (logical expression) then
statements
:
:
else
statements
endif
```

Loops

For repeated execution of similar things, loops are used. Fortran 77 has only one loop construct, called the do-loop.

do-loops
The do-loop is used for simple counting. Here is a simple example that prints the cumulative sums of the integers from 1 through n (assume n has been assigned a value elsewhere):

```
          integer i, n, sum
            sum = 0
           do 10 i = 1, n
               sum = sum + i
                  write(*,*) 'i =', i
10          continue
          write(*,*) 'sum =', sum
```

The number 10 is a statement label. The numerical value of statement labels have no significance, so any integer numbers can be used.
The variable defined in the do-statement is incremented by 1 by default. However, you can define any other integer to be the step. This program segment prints the even numbers between 1 and 10 in decreasing order:

```
         integer i
        do 20 i = 10, 1, -2
           write(*,*) 'i =', i
20       continue
```

The general form of the do loop is as follows:

```
          do label ivar = e1, e2, e3
            statements
label       continue
```

ivar is the loop variable (often called the loop index) which must be integer. e1 specifies the initial value of *ivar*, e2 is the terminating bound, and e3 is the increment (step).

A DO loop is executed as follows:

1. The expressions e1, e2 and e3 are evaluated. Since ivar is integer, e1, e2 etc. are converted to integer type. If e3 is omitted, a default value of 1 is used. The resulting values are called the parameters of the loop. We shall call them initial, limit and increment respectively.

2. initial is assigned as a value to *ivar*.

3. *ivar* is compared with limit, the test depending on the value of increment as follows:
If increment > 0 test whether *ivar* $\leq$ limit
If increment < 0 test whether *ivar* $\geq$ limit
If the condition tested is "true", then:
(i) the range of the DO loop is executed,
(ii) *ivar* is incremented by increment,
(iii) control returns to step 3.

Otherwise: iteration stops and execution continues with the statement following the terminal statement.

Some examples:

DO 10 I = 1,5
causes the range of statements beginning with the next and ending with the statement labelled 10 to be executed 5 times.

DO 10 I = 0,100,5
causes the range to be executed 21 times for values of I of 0,5,10...100.

DO 10 I = 100,0,-5
causes the range to be executed 21 times for values of I of 100,95...0.

DO 10 I = 0,100,-5
In this case, the range is not executed at all, as the test in step 3 fails for the initial value of I.

Restrictions and other notes
The do-loop variable must **never** be changed by other statements within the loop! This will cause great confusion.
Other rules:
1. Increment must not be zero.
2. The range of a DO loop can be entered only via the initial DO statement. Thus a GOTO cannot cause a jump into the range of a DO loop. However, GOTOs can be included in the range to jump to statements either inside or outside it. In the latter case, this can cause iteration to stop before the control variable reaches the limiting value.

Examples:

```
                GOTO 10
                . . .
                DO 20, I = 1,5
10      . . .
20      CONTINUE
```

is wrong, but

```
                DO 20, I = 1,5
10      . . .
                IF (...) GOTO 10
                IF (...) GOTO 30
20      CONTINUE
                . . .
30      . . .
```

is all right.

3. The loop parameters are the values of the expressions e1, e2 and e3 on entry to the loop. The expressions themselves are not used. Therefore if any of e1, e2 and e3 are variables, they can be assigned values within the loop without disrupting its execution, but should be avoided if possible.

Important:
The control variable is incremented and tested at the end of each iteration. Thus, unless iteration is interrupted by a GOTO, the value of the control variable after execution of the loop will be the value which it was assigned at the end of the final iteration. Example: in the following loop
DO 10, I = 0,100,5
the control variable I is incremented to exactly 100 at the end of the 20th iteration. This does not exceed limit, so another iteration is performed. I is then incremented to 105 and iteration stops, with I retaining this value.

do and enddo

do-loops can be closed by the enddo statement instead of a continue statement and a statement label can then be omitted.

```
do i = 1,10
.....
enddo
```

while-loops

General form	Example
do while (logical expr)	do while (i.lt.5)
statement	y = y+1
statement	i = i+1
enddo	enddo

do can occur in the following form also: (infinite loop terminated by a condition)

```
do
If (i.eq.j) cycle
...
if (i > j) exit
...
enddo
```

The if statement enables the program to get out of the infinite loop.
Here if i = j, the statements following the if statement will not be executed and the loop will be continued.
Exit is a general command by which one can get out of any do loop.
Nested loops:

Valid

```
      DO 20 ...
        ...
      DO 10 ...
        ...
10    CONTINUE
        ...
20    CONTINUE
```

Invalid

```
      DO 20 ...
        ...
      DO 10 ...
        ...
20    CONTINUE
        ...
10    CONTINUE
```

Arrays

Take the example of calculating the dot product of two three dimensional vectors. It is easy to compute: dot product = $a_1b_1 + a_2b_2 + a_3b_3$. While writing a program, you have to store the components as $a1, a2$ etc. The program can be written in one line.
But suppose I now increase the dimension of the vector, to say, 27, it becomes too clumsy; we have to use 54 variable names and one long line for the calculation.
One way to avoid this is to use an **array** : instead of a1, a2, ...a27 we now have a single **subscripted variable** a, where it is understood that a(1), a(2)....etc. represent the components.
Then the program can be done like this:

```
sum =0.0
do i = 1, 27
sum = sum + a(i)*b(i)
enddo
```

Of course this is not the whole story!

First let us use the formal definitions:

Fortran uses a structure similar to a vector called an array. An array A with N elements is an ordered list of N variables of a given type, called the elements of the array. In Fortran, the subscript notation used for the components of a vector is not available. Instead the elements are denoted by the name of the array followed by an integer expression in parentheses. Thus, the elements of A are denoted by A(1), A(2),... A(N). The parenthesised expressions are called array subscripts even though not written as such.
A subscript can be any arithmetic expression which evaluates to an integer. Thus, if A, B, and C are arrays, the following are valid ways of writing an array element:

```
A(10)
B(I+4)
C(3*I+K)
```

Array declarations

Since an array is a list of variables, it obviously requires several words or other units of storage. Each array must therefore be *declared* in a statement which tells the compiler how many units to reserve for it. This can be done by including the array name in a type specification followed by its dimension in parentheses. For example:

```
INTEGER AGE(100),NUM(25),DEG
```

This reserves 100 words of storage for array AGE, 25 for array NUM, and one word for the variable DEG. All three items are of type INTEGER.
Space can also be reserved for arrays by the DIMENSION statement, which reserves storage using a similar syntax, but includes no information about type. Thus, if this method is used, the type is either determined by the initial letter of the array or assigned by a separate type specification. Therefore, the equivalent to the above using a DIMENSION statement is:

```
INTEGER AGE,DEG
DIMENSION AGE(100),NUM(25)
```

(NUM is treated as INTEGER by default).

DIMENSION statements, like type specifications, are non-executable and must be placed before the first executable statement.

When this form of declaration is used in a type or DIMENSION statement the upper and lower bounds for the subscript are 1 and the dimension respectively. Thus, AGE in the above example may have any subscript from 1 to 100. Arrays can also be declared to have subscripts with a lower bound other than 1 by using a second form of declaration in which the lower and upper bounds are given, separated by a colon. For example:

```
REAL C(0:20)
INTEGER ERROR(-10:10)
```

reserves 21 words of storage for each of the arrays C and ERROR and stipulates that the subscripts of C range from 0 to 20 inclusive, while those of ERROR range from -10 to 10.

Caution

Although the declaration stipulates bounds for the subscript, not all compilers check that a subscript actually lies within the bounds. For example, if NUM is declared as above to have a subscript from 1 to 25, a reference to NUM(30) may not cause an error. The compiler may simply use the 30th word of storage starting from the address of NUM(1) even though this is outside the bounds of the array.

This can cause unpredictable results. Care should therefore be taken to make sure that your subscripts are within their bounds.

Use of arrays and array elements

Array elements can be used in the same way as variables, their advantage being that different elements of an array can be referenced by using a variable as a subscript and altering its value, for example by making it the control variable of a DO loop.
The array name without a subscript refers to the entire array and can be used only in a number of specific ways.

Initialising an array

Values can be assigned to the elements of an array by assignment statements, e.g.

```
      NUM(1) = 0
      NUM(2) = 5
```

If all the elements are to have equal values, or if their values form a regular sequence, a DO loop can be used. Thus, if NUM and DIST are arrays of dimension 5:

```
        DO 10 I = 1,5
        NUM(I) = 0
10      CONTINUE
```

initialises all the elements of NUM to 0, while:

```
      DO 10 I = 1,5
      DIST(I) = 1.5*I
10    CONTINUE
```

initialises the values DIST(1) = 1.5, DIST(2) = 3.0 etc.
Current compilers also allow

```
      A = 0
```

which assigns all the elements of A a zero value.

Input and output statements: Implied do loops

Read statements for subscripted variables may be of the following forms:

```
      Read (*,*) (a(i), i=1, 10)
```

Reads data given either in a line or in a line by line manner. Data are read as a(1), a(2) ...

```
      do i = 1,10
      Read (*,*) a(i)
      enddo
```

Reads data given as one data per line.

Write statements are of the following forms:

```
write (*,*) (a(i), i=1, 10)
```

Writes data in a single line as a(1), a(2)....

```
do i = 1,10
write (*,*) a(i)
enddo
```

Prints one data per line.
Write statement with format:

```
write (*,45) (a(i), i=1, 3)
45   format (3(e15.7,2x))
```

Arrays of higher dimension:

Arrays can be multidimensional. A two dimensional array A will have elements typically written as A(I,J) with a corresponding declaration statement:

```
Real A(10,100)
```

which means the first index in A can run from 1 to 10 and the second from 1 to 100. Total number of elements is 10 X 100 = 1000. A(I,J) can be easily identified as the elements of a matrix.

Read/write statements in the implied do loop form:

```
Real a(10,100)
n = 10
m = 20
Read (*,*) ((a(i,j), j=1,n),i=1,m)
write (*,*) ((a(i,j), j=1,n),i=1,m)
```

This will mean that the data is read in the order a(1,2), a(1,3), a(1,4).... a(1,10), a(2,1),a(2,2).....a(2,10),.....a(20,1), a(20,2).....a(20,10).
Similarly for the write statement.

But you may want to print a matrix in a matrix form.
Proper way to do it:

```
do 10 i = 1, 20
        do 10 j = 1, 10
         write (*,*) (a(i,j), j=1,10)
         enddo
 enddo
```

DATA statement

The DATA statement is a non-executable statement used to initialise variables. An example
DATA A,B,N/1.0,2.0,17/
A constant may be repeated by preceding it by the number of repetitions required (an integer) and an asterisk. Thus:
DATA N1,N2,N3,N4/4*0/
means N1, N2, N3 and N4 are assigned a value 4.0

It is particularly useful for initialising arrays.
For example if A is an array of dimension 20, the DATA statement:
DATA A(1),A(2),A(3),A(4)/4*0.0/,A(20)/-1.0/
assigns a value of zero to the first four elements, -1.0 to the last element, and leaves the remaining elements undefined.

A PARAMETER statement is used to assign a constant value to a symbolic name. Example
PARAMETER(L = 100, x = 2.5, N=1000)
Although the value of a FORTRAN constant cannot be changed elsewhere in the program, it can be used in other PARAMETER statements as well as in type declarations, DATA statements and in calculations.
PARAMETER(NCOMP=12)
REAL VECTOR(NCOMP)
DATA N/NCOMP/

SINGLE PRECISION and DOUBLE PRECISION data
GNU Fortran uses up to a specific number (say n) of digits to represent real numbers. If e.g., n= 10, it will report that
sqrt (3.) = 1.73205078 , sqrt (1100.) = 33.1662483 , sqrt (2.25) = 1.5 .
In working in single precision it is futile to assign more than n nonzero digits to represent a number, as Fortran will change all further digits to 0. The assignments
x = 123456789876543. , x = 123456789800000. , x = 1234567898E5
for n=10 produce the same result if x already has been declared a single precision real number.

A number stored in a computer is limited in magnitude and precision. The limits depend on the particular computer. Thus, a REAL number has only a certain number of significant digits. If more significant digits are required for a calculation, then DOUBLE PRECISION numbers must be used. A DOUBLE PRECISION constant is written in the same exponential form as a single precision REAL constant except with a D instead of an E separating the mantissa from the exponent (same in FORMAT statements).

In practice, most computers use 32 bits to store INTEGER and REAL numbers. This means that an INTEGER is limited to numbers between -2,147,483,648 and +2,147,483,647 (a sign bit and 31 magnitude bits). If the IEEE standard is used, then a REAL number will have about seven decimal digits and be within the magnitude range of 10-38 to 10+38. DOUBLE PRECISION numbers usually have at least twice the number of significant decimal digits but may have the same magnitude range of REAL numbers.
Some ways of writing the number 12.345 as a double precision real number are
1.2345D1 , .12345D2 , .012345D3 , 12.345D0 , 12345D-3 .

When assigning a value to a double precision variable you should use this D-scientific notation, as otherwise the value will be read only in single precision. For example, if A is double precision and you want to assign A the value 3.2, you should write
A = 3.2D0
instead of just A = 3.2.

When a number is input from the keyboard in response to a read (*,*) command, the user need not worry about types or input format. Suppose for example that x is single or double precision, and the user is to enter a value for x in response to the command read (*,*) x. If the user enters simply "3" (integer format), GNU Fortran will change 3 to the proper format (to 3. if x is single precision and to 3D0 if x is double precision) before assigning it to x. Likewise, if x is double precision and the user enters 3.1 (single precision format), Fortran converts 3.1 to 3.1D0 before assigning it to x. (However, with an ordinary assignment statement x = 3.1 from within the program, the number is not changed to double precision format before being assigned to x.)
A number x can be converted to double precision by the function dble(x)

Declaration statement for double precision variable A is simply
DOUBLE PRECISION A
at the beginning of a program
A program that has only DOUBLE PRECISION variables might contain the statement
IMPLICIT DOUBLE PRECISION(A-Z)
at the beginning.

Statement function and subprograms

A function may be defined as a statement in the fortran program.
Examples

f(x) = cos(x)-exp(-a*x)
root(a,b,c) = (-b + sqrt(b**2-4.0*a*c))/(2.0*a)

A statement function must appear only after the declaration statements and before the first executable statement of the program unit in which it is referenced.
A statement function is not executed at the point where it is defined. It is executed by refereeing to the function in expressions/assignments used in the program.
For example:

```
f(x) = cos(x) - exp(-a*x)
a= 5.0
y = 1/f(x)
 .........
```

Here, in the first line the function is defined and in the third line, the function is evaluated using the specified value of a.

Subprograms:

There are two types of subprograms: Function and Subroutine

Function subprogram: The purpose of a function subprogram is to take in a number of values or arguments, do some calculations with those arguments and then return a *single* result. Library functions are such programs written into fortran and can be used without any special effort by the programmer.
The general way to activate a function is to use the function name in an expression. The function name is followed by a list of inputs, also called arguments, enclosed in parenthesis:

```
Real Function average(x,y,z)
Real x,y,z
average = (x+y+z)/3.0
return
end
```

In the main program, the function is used in the following way

```
c         Main program
          real a,b,c, average
          Read (*,*) a,b,c
          av = average(a,b,c)
          write (*,*) av
          .....
```

There is a one to one correspondence between x,y,z and a,b,c. The type declarations must match for the function in the main program and subprogram.

Subroutine Subprogram Function subprograms yield only one return value, but one might need to obtain more than one result from some input values (e.g. sum, average, sum of the squares etc.)
For such a case, one needs to write a subroutine subprogram. Example:

```
Subroutine CALC(x,y,z,sum,sumsq,average)
real x, y, z, sum, sumsq, average
sum = x+ y + z
sumsq = x*x + y*y +z*z
average = (x+y+z)/3.0
return
end
```

CALC appears only as the name of the subroutine and does not represent a variable name. The way the subroutine is used in the main program is as follows:

```
c         Main program
          real a,b,c, sumabc, sumsq, av_abc
          read (*,*) a,b,c
          call CALC(a,b,c,sumabc, sumsq, av_abc)
          write (*,*) a,b,c, sumabc, sumsq, av_abc
          ......
```

Once again there is one to one correspondence between the arguments of the subroutine in the main and the subprogram. There has to be type matching also. Check that both inputs and outputs all appear as arguments of the subroutine.

One can use arrays in subprograms also.
An example: let A(i) be the elements of an array for which one needs to compute the average and the smallest value of A. The following subroutine may be used

```
Subroutine values(A,n,aver,small)
integer n
dimension A(n)
real small, sum, aver
sum = 0.0
small = A(1)
do i = 1,n
    sum = sum + A(i)
    if (i.gt.1) then
       if (A(i).lt.small) small = A(i)
    endif
enddo
aver = sum/float(n)
return
end
```

In the main program we have

```
c          Main program
           dimension B(100)
           read (*,*) (B(i), i = 1, 10)
           call values(B,10,bsmall, baver)
           write  (*,*)  bsmall, baver
           ......
```

Similarly for the function subprogram.

One can refer to a function in a subroutine and vice versa.
One can also use a function common to the main program and the subroutine.
An example:

```
c Integration program which uses external function from
c function subprogram
c alim_1 and alim_2 limits of the integral
c two different upper limits are used

        external f
        h= 0.01
        alim_1 = 0.
        alim_2 = 1.
        val1 = 0.
        write (*,*) 'lower limit', alim_1
74      call simp(f,alim_1,alim_2,h,val1)
        write (*,*) 'upper limit', alim_2, val1
        alim_2 = 2.0
75      call simp(f,alim_1,alim_2,h,val2)
        write (*,*) 'upper limit', alim_2, val2
        end

        subroutine simp(f,alim_1,alim_2,h,val)
c       function is evaluated from function subproram
        x1= alim_1
        x2 = alim_2
        iter = nint((alim_2  - alim_1)/h)
        sum = f(x1) + f(x2)
        a1 = 0.
        a2 = 0.
        do i = 1,iter,2
                a1 = a1+ f(x1+i*h)
                if (i.eq.(iter-1)) go to 45
                a2 = a2 + f(x1+(i+1)*h)
        enddo
45      val = h*(4.*a1+2.*a2)/3. + sum*h/3.
        return
        end

        function f(x)
        f = x**3
        return
        end
```

Common blocks

Fortran 77 has no global variables, i.e. variables that are shared among several program units (subroutines). The only way to pass information between subroutines we have seen so far is to use the subroutine parameter list. Sometimes this is inconvenient, e.g., when many subroutines share a large set of parameters. In such cases one can use a common block. This is a way to specify that certain variables should be shared among certain subroutines. But in general, the use of common blocks should be minimized.

Example:

Suppose you have two parameters alpha and beta that many of your subroutines need. The following example shows how it can be done using common blocks.

```
      program main
      real alpha, beta
      common /coeff/ alpha, beta
      ........
      stop
      end

      subroutine sub1(some arguments)
c      declarations of arguments
      real alpha, beta
      common /coeff/ alpha, beta
      .........
      return
      end

      subroutine sub2(some arguments)
c      declarations of arguments
      real alpha, beta
      common /coeff/ alpha, beta
      .......
      return
      end
```

Here we define a common block with the name coeff. The contents of the common block are the two variables alpha and beta. A common block can contain as many variables as you like. They do not need to all have the same type. Every subroutine that wants to use any of the variables in the common block has to declare the whole block.
Note that in this example we could easily have avoided common blocks by passing alpha and beta as parameters (arguments). A good rule is to try to avoid common blocks if possible. However, there are a few cases where there is no other solution.
Genral syntax

common / name / list-of-variables

The common statement should appear together with the variable declarations, before the executable statements. Different common blocks must have different names (just like variables). A variable cannot belong to more than one common block. The variables in a common block do not need to have the same names each place they occur (although it is a good idea to do so), but they must be listed in the same order and have the same type and size.
To illustrate this, look at the following continuation of our example:

```
subroutine sub3 (some arguments)
declarations of arguments
real a, b
common /coeff/ a, b

statements
return
end
```

This declaration is equivalent to the previous version that used alpha and beta.