Programming Basics - FORTRAN 77

http://www.physics.nau.edu/~bowman/PHY520/F77tutor/tutorial_77.html

Program Organization

A FORTRAN program is just a sequence of lines of plain text. This is called the *source code*. The text has to follow certain rules (syntax) to be a valid FORTRAN program. We start by looking at a simple example:

```
      program circle
      real r, area, pi

c This program reads a real number r and prints
c the area of a circle with radius r.

      write (*,*) 'Give radius r:'
      read  (*,*) r
      pi = atan(1.0e0)*4.0e0
      area = pi*r*r
      write (*,*) 'Area = ', area

      end
```

A FORTRAN program generally consists of a main program and possibly several subprograms (*i.e.*, functions or subroutines). The structure of a main program is:

```
      program name

      declarations

      statements

      end
```

Note: Words that are in italics should not be taken as literal text, but rather as a description of what belongs in their place. FORTRAN is not case-sensitive, so "X" and "x" are the same variable. Blank spaces are ignored in Fortran 77. If you remove all blanks in a Fortran 77 program, the program is still acceptable to a compiler but almost unreadable to humans.

Column position rules

Fortran 77 is *not* a free-format language, but has a very strict set of rules for how the source code should be formatted. The most important rules are the column position rules:

```
Col. 1:    Blank, or a "c" or "*" for comments
Col. 1-5:  Blank or statement label
Col. 6:    Blank or a "+" for continuation of previous line
Col. 7-72: Statements
```

Comments

The lines that begin with a "c" or "*" are comments and have no purpose other than to make the program more readable for humans.

Continuation

Sometimes, a statement does not fit into the 66 available columns of a single line. One can then break the statement into two or more lines, and use the continuation mark in position 6. Example:

```
c The next statement goes over two physical lines
      area = 3.14159265358979
     +       * r * r
```

Any character can be used instead of the plus sign as a continuation character. It is considered good programming style to use either the plus sign or digits (using 2 for the second line, 3 for the third, and so on).

Declarations

Every variable should be defined in a declaration. This establishes the type of the variable. The most common declarations are:

```
      integer   list of variables
      real      list of variables
      logical   list of variables
      character list of variables
```

The list of variables should consist of variable names separated by commas. Each variable should be declared exactly once.

The `parameter` statement

Some constants appear many times in a program. It is then often desirable to define them only once, in the beginning of the program. This is what the parameter statement is for. It also makes programs more readable. For example, the circle area program should rather have been written like this:

```
      program circle
      real r, area, pi
      parameter (pi = 3.14159)

c This program reads a real number r and prints
c the area of a circle with radius r.

      write (*,*) 'Give radius r:'
      read  (*,*) r
      area = pi*r*r
      write (*,*) 'Area = ', area

      end
```

The syntax of the parameter statement is

```
      parameter (name = constant, ... , name = constant)
```

Assignment

A variable assignment has the form:

```
variable_name = expression
```

The interpretation is as follows: Evaluate the right hand side and assign the resulting value to the variable on the left. The expression on the right may contain other variables, but the variable assignment does not change their value. The following example does not change the value of pi or r, only area:

```
area = pi*r*r
```

Logical expressions

Logical expressions can only have the value .TRUE. or .FALSE. A logical expression can be formed by comparing arithmetic expressions using the following relational operators:

```
.LT.  meaning <
.LE.          <=
.GT.          >
.GE.          >=
.EQ.          =
.NE.          /=
```

The `if` statement

An important part of any programming language is the conditional statements. The most common such statement in FORTRAN is the if statement. The following is a simple example of the logical if statement that finds the absolute value of x:

```
if (x .LT. 0) x = -x
```

The most general form of the if statement has the following form:

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

The execution flow is from top to bottom. The conditional expressions are evaluated in sequence until one is found to be true. Then the associated statements are executed and the control resumes after the endif.

Loops

For repeated execution of similar things, *loops* are used. The do-loop is used for simple counting. Here is a simple example that prints the cumulative sums of the integers 1 through 10:

```
integer i, n, sum

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

The number 10 is a *statement label*. Typically, there will be many loops and other statements in a single program that require a statement label. The programmer is responsible for assigning a unique number to each label in each program (or subprogram). Recall that column positions 1-5 are reserved for statement labels. The numerical value of statement labels has no significance, so any integers can be used, in any order. Typically, most programmers use consecutive multiples of 10.

Arrays

Many scientific computations use vectors and matrices. The data type FORTRAN uses for representing such objects is the array. A one-dimensional array corresponds to a vector, while a two-dimensional array corresponds to a matrix. The simplest array is the one-dimensional array, which is just a sequence of elements stored consecutively in memory. For example, the declaration

```
real vect1(20)
```

declares `vect1` as a real array of length 20. By convention, FORTRAN arrays are indexed from 1 and up. Thus the first number in the array is denoted by `vect1(1)` and the last by `vect1(20)`. However, you may define an arbitrary index range for your arrays using the following syntax:

```
real b(0:19), weird(-162:237)
```

Here, `b` is similar to `vect1` from the previous example, except the index runs from 0 through 19, while `weird` is an array of length 237-(-162)+1 = 400. Each element of an array can be thought of as a separate variable. You reference the i'th element of array `a` by `a(i)`. Here is a code segment that stores the squares of the numbers 1 through 10 in the array `sq`:

```
integer i, sq(10)

do i = 1, 10
   sq(i) = i**2
enddo
```

A common bug in Fortran is that the program tries to access array elements that are out of bounds or undefined. This is the responsibility of the programmer, and the Fortran compiler will not detect any such bugs!

Matrices are very important in linear algebra. Matrices are usually represented by two-dimensional arrays. For example, the declaration

```
real A(3,5)
```

defines a two-dimensional array of 3*5=15 real numbers. It is useful to think of the first index as the row index, and the second as the column index. Hence we get the graphical picture:

```
   (1,1)  (1,2)  (1,3)  (1,4)  (1,5)
   (2,1)  (2,2)  (2,3)  (2,4)  (2,5)
   (3,1)  (3,2)  (3,3)  (3,4)  (3,5)
```

It is quite common in Fortran to declare arrays that are larger than the matrix we want to store. (This is because Fortran does not have dynamic storage allocation.) This is perfectly legal. Example:

```
      real A(3,5)
      integer i,j
c
c     We will only use the upper 3 by 3 part of this array.
c
      do j = 1, 3
         do i = 1, 3
            A(i,j) = real(i)/real(j)
         enddo
      enddo
```

The elements in the submatrix A(1:3,4:5) are undefined. Do not assume these elements are initialized to zero by the compiler (some compilers will do this, but not all).

Functions

FORTRAN functions are quite similar to mathematical functions. They both take a set of input arguments (variables) and return a value of some type. Fortran 77 has some intrinsic (built-in) functions. The following example illustrates how to use a function:

```
      x = cos(pi/3.0)
```

Here cos is the cosine function, so x will be assigned the value 0.5 assuming pi has been correctly defined; Fortran 77 has no built-in constants. There are many intrinsic functions in Fortran 77. Some of the most common are:

`abs`	*absolute value*
`sqrt`	*square root*
`sin`	*sine*
`tan`	*tangent*
`atan`	*arctangent*
`exp`	*exponential (natural)*
`log`	*logarithm (natural)*

Input/Output

An important part of any computer program is the handling of input and output. In our examples so far, we have already used the two most common Fortran constructs for this: `read` and `write`. Fortran I/O can be quite complicated, so we will only describe some simpler cases in this handout. `Read` is used for input, while `write` is used for output. A simple form is

```
      read (unit no, format no) list-of-variables
      write(unit no, format no) list-of-variables
```

The unit number can refer to either standard input, standard output, or a file. The format number refers to a label for a format statement. It is possible to simplify these statements further by using asterisks (*) for some arguments, like we have done in most of our examples so far. This is sometimes called *list directed* read/write.

```
      read (*,*) list-of-variables
      write(*,*) list-of-variables
```

The first statement will read values from the standard input and assign the values to the variables in the variable list, while the second one writes to the standard output. It is also possible to read from or write to *files*. Before you can use a file you have to *open* it. The command is

```
    open (list-of-specifiers)
```

The most common specifiers are:

```
    unit = u
    file = filename
```

where *u* is the unit number in the range 1-99 that denotes this file (the programmer may chose any number but he/she has to make sure it is unique) and *filename* is a character string denoting the file name. After a file has been opened, you can access it by read and write statements. When you are done with the file, it should be closed by the statement

```
      close (list-of-specifiers)
```

For example, you are given a data file with xyz coordinates for a bunch of points. The number of points is given on the first line. The file name of the data file is *points.dat*. Here is a short program that reads the data into 3 arrays x,y,z:

```
      Program inpdat
c
c  This program reads n points from a data file and stores them in
c  3 arrays x, y, z.
c
      integer nmax
      parameter (nmax=1000)
      real x(nmax), y(nmax), z(nmax)

c  Open the data file
      open (unit=20, file='points.dat')

c  Read the number of points
      read(20,*) n
      if (n.GT.nmax) then
         write(*,*) 'Error: n = ', n, 'is larger than nmax =', nmax
         goto 9999
      endif

c  Loop over the data points
      do i= 1, n
         read(20,100) x(i), y(i), z(i)
      enddo
 100 format (3(F10.4))

c  Close the file
      close (20)

c  Now we can process the data somehow...
 9999 stop
      end
```

Subroutines

When a program is more than a few hundred lines long, it gets hard to follow. FORTRAN codes that solve real problems often have tens of thousands of lines. The only way to handle such big codes is to use a *modular* approach and split the program into many separate smaller units called *subroutines*. The syntax is very similar to the structure of the main program except that it is identified as a subroutine and there is a return statement before the end. The following is an example of a simple subroutine used to swap two integers.

```
      subroutine iswap (a,b)

c input/output variables
      integer a, b

c local variables
      integer tmp

      tmp = a
      a = b
      b = tmp

      return
      end
```

Note that there are two blocks of variable declarations here. First, we declare the input/output parameters, i.e. the variables that are common to both the caller and the callee. Then we declare the *local variables*, i.e. the variables that can only be used within this subprogram. We can use the same variable names in different subroutines and the compiler will know that they are different variables that just happen to have the same names. Subroutines are invoked using the word *call* before their names and parameters.

```
      program callex
      integer m, n
c
      m = 1
      n = 2

      call iswap(m,n)
      write(*,*) m, n

      end
```

Creating an Executable

When you have written a FORTRAN program, you should save it in a file that has the extension `.f` Before you can execute the program, you need to translate source code into machine readable form. This is done by a special program called the *compiler*. The Unix command that typically runs the Fortran 77 compiler is `f77`. (Note: The FORTRAN compiler used on cgate, however, is invoked by the command `xlf`.) The compiler translates source code into *object code* and the linker/loader makes this into an *executable*. The default output from the compilation is given the somewhat cryptic name `a.out`, but you can choose another name if you wish using the -o option. For example,

```
    f77 circle.f -o circle.out
```

will compile the file *circle.f* and save the executable in the file *circle.out* (rather than the default *a.out*).

In the previous example, we have not distinguished between *compiling* and *linking*. These are two different processes but the FORTRAN compiler performs them both, so the user usually does not need to know about it. But in the next example we will use *two* source code files.

```
f77 circle1.f circle2.f
```

This will generate *three* files, the two object code files *circle1.o* and *circle2.o*, plus the executable file *a.out*. What really happened here, is that the Fortran compiler first compiled each of the source code files into object files (ending in .o) and then linked the two object files together into the executable *a.out*. You can separate these two steps by using the -c option to tell the compiler to only compile the source files:

```
f77 -c circle1.f circle2.f

f77 circle1.o circle2.o
```

Compiling separate files like this may be useful if there are many files and only a few of them need to be recompiled. You can imagine that a program containing multiple source files would take a lot of typing to compile. In addition, if you have enough lines of source code, it can take a long time for the compiler to actually compile all of them.   In UNIX there is a useful program which will let you automatically compile all of your source files by simply typing the UNIX command `make`. Furthermore, if you've only changed some of the files, it will only recompile those source files which depend upon files in which you have made changes.   Of course, you have to tell `make` which files depend on which other files by creating a file called Makefile. Large software packages with many source files typically come with a *makefile* and all the user has to do to create an executable is simply type *make*.