Introduction to programming in Fortran 77 for students of Science and Engineering

Roman Gröger

University of Pennsylvania, Department of Materials Science and Engineering
3231 Walnut Street, Office #215, Philadelphia, PA 19104

Revision 1.2 (September 27, 2004)

---

1 Introduction

Fortran (FORmula TRANslation) is a programming language designed specifically for scientists and engineers. For the past 30 years Fortran has been used for such projects as the design of bridges and aeroplane structures, it is used for factory automation control, for storm drainage design, analysis of scientific data and so on. Throughout the life of this language, groups of users have written libraries of useful standard Fortran programs. These programs can be borrowed and used by other people who wish to take advantage of the expertise and experience of the authors, in a similar way in which a book is borrowed from a library.

Fortran belongs to a class of higher-level programming languages in which the programs are not written directly in the machine code but instead in an artifical, human-readable language. This *source code* consists of *algorithms* built using a set of standard *constructions*, each consisting of a series of *commands* which define the elementary operations with your data. In other words, any algorithm is a cookbook which specifies input ingredients, operations with them and with other data and finally returns one or more results, depending on the function of this algorithm. Any source code has to be *compiled* in order to obtain an executable code which can be run on your computer. The compilation is performed by a separate program, called *compiler*, which translates your text-based source code into the machine code that can be directly interpreted by your computer processor.

The task of writing a program involves several steps which are common to all programming languages:

- **problem specification** – each program has a motivation to be written. This is usually represented by a particular assignment which should clearly state what is the purpose of writing your code.

- **analysis of the problem** – this is a very important step prior to writing a new code, which is often omitted by beginners. When writing more complicated algorithms, you should always start on a paper. If the algorithm has to solve a particular mathematical problem, carry out the derivation or expression of a particular unknown on the paper. Be careful to check the physical dimensions of the input and output parameters of your function.

- **writing the source code** – once finished with the analysis of your problem, you can write a source code of this algorithm. Fortran provides you with a number of *intrinsic functions*, mostly standard mathematical operations like square root, sine, cosine, exponential function or logarithm, which you can directly use in your code. Any other mathematical functions can be written as separate subprograms using a set of standard arithmetic operations. You can thus build a *library* of the most useful mathematical functions and simply call them any time you need them.

- **compiling the code** – means the same as translating your source code from the language of Fortran to the language of your computer. Compilation produces an executable code which can be subsequently interpreted in the processor of your computer. Have you ever tried to display the contents of an `.EXE` file under Windows ? If so, then you know what the machine code really looks like.

- **running and testing the program** – although your program may be running without apparent errors, it can have a number of hidden *bugs*. Take some time to play with the running code and test if it really does what it should. It frequently happens that you mistype some arithmetic operation in your source code, which in turn might give you totally different results. Never believe that you write a clever program without a detailed testing.

To write programs in Fortran, you will need a good editor which allows you to type your source code. Many simple editors like `Notepad` under Windows or `pico`, `nano` under Linux do not offer you the functionality that you will certainly need for writing larger codes. Good text editor should allow you to display your code in color so that comments, identifiers, variables and commands are distinguished from each other. Moreover, because F77 imposes special requirements on the indentation of some parts of your source code, it is useful to have an editor which can, usually after pressing `Tab`, automatically set the cursor at the position where you have to start writing your text. I strongly recommend you to use `emacs` which is an intelligent programmer's editor that has all the features you can imagine. More importantly, it exists on many platforms (Windows, MacOS, Linux, ...) and therefore once you become familiar with using it on one platform, you can equally well type your text under Linux or on the Mac in your lab.

In contrast, the compilation of your code is a step which is strongly platform-dependent which means that the executable codes are not transferrable between different operating systems. This is not so bad as it may look for the first sight. If your friend needs to run your code on Mac, whereas you work with Windows, simply give them your source code and ask them to compile it on their computer. Each platform you may come in contact with is nowadays equipped with a good F77 compiler, some of which are even distributed for free. In this course, we will use `g77` (GNU Fortran) compiler which can be downloaded from the Internet for both Windows and MacOS. If you intend to use Linux, this compiler is most likely embedded in your distribution.

The communication between you, a programmer, and your computer will always occur via the *command-line terminal*. If you expected that you will learn some special windows-based application for writing and compiling F77 codes, you are now probably a little disappointed. However, you will shortly understand that using the command-line allows you to focus on mere writing your code, rather than on fighting with a new graphical application. Once you get into using your command-line on one platform, it will be very simple for you to use any other operating system.

This document will guide you through download, installation and setup of everything what is needed to get started with programming on your computer. During this journey, operations which differ under Windows and MacOS X are grouped under different icons. Please note that the installation procedure for MacOS is devoted to version 10 which contains a Linux command-line terminal. Because the target audience of this text are mainly undergraduate students who do not have any previous knowledge of programming, each problem will be treated in the simplest and most straightforward way. Once you become familiar with programming in F77, you will quickly find out that the same problem can be solved many different ways.

2 Setting up your command-line terminal

Setup of your command-line terminal is a crucial step which allows you to gain control over the communication with your computer. It will be useful for you to place a terminal icon (or an alias under MacOS) on your desktop, so that the command-line terminal will be quickly accessible when needed. From the paragraphs below, choose the operating system which you use on your computer.

To create a shortcut of the command-line terminal, click on the desktop with the right mouse button, choose `New` and then `Shortcut`. In the input line, write `cmd` and click on the button `Next`. If nothing happens (mainly in older versions of Windows), try writing `command` instead. In the next window, assign this icon the name `Terminal` and click on `Finish`. Your desktop should now display a shortcut for your command-line terminal. Open up your terminal window.

If you are a lucky owner or user of Apple with running MacOS X, the setup of your command-line terminal is a straightforward task. Click on the `hard drive` icon which should be on your desktop, then on `Applications` in the upper part of the window and open the folder `Utilities`. You should now see the icon `Terminal`. To create its alias, press and hold `Ctrl` and click on this icon, then choose `Make alias`. Finally drag the icon `Terminal alias` onto the desktop and, possibly, give it a name `Terminal`. Open up your terminal window.

3 Download and setup

Make sure that the command-line terminal is open on your desktop. We now proceed with downloading and setting up the `emacs` editor and the `g77` compiler.

Follow these steps to download Emacs for Windows 95/98/NT/2k/Me/XP. All steps must be performed *exactly* as written below, otherwise you may not be able to run Emacs.

1. Pick the drive and a folder in which you want to install Emacs. We will assume that it is `c:\emacs`, but you can choose a different one. If you choose a different drive or a different folder, you will need to adapt the directions below accordingly.

2. Download the Power Archiver for Windows from `http://www.powerarchiver.com`. This program will help you to decompress the files with `.tar.gz` extension. Install the program.

3. Find `ftp://ftp.gnu.org/gnu/windows/emacs/` and download `emacs-*-bin-i386.tar.gz` to `c:\emacs.tar.gz` by right-clicking on the link. Make sure the entire file was downloaded without any network errors. Use `Save Link As...` or `Save Target As...` This file is a compressed distribution of `emacs` for Windows.

4. Go to `http://www.geocities.com/Athens/Olympus/5564/g77.htm` and find the links to files `g77exe.zip` and `g77lib.zip`. Save both of them as `c:\g77exe.zip` and `c:\g77lib.zip` by using `Save Link As...` or `Save Target As...` The former file is a compressed distribution of `g77` Fortran compiler for Windows and the latter one is the set of standard Fortran libraries.

5. Check the `c:\` folder to make sure that it contains files `emacs.tar.gz`, `g77exe.zip` and `g77lib.zip`. Uncompress all of them using the Power Archiver which you downloaded above.

6. Check the `c:\` folder to make sure that it contains subfolders `emacs-*`, with the asterisk replacing the version number, and `g77`. To make our life a little easier, rename the Emacs folder by typing the command
   ```
   rename emacs-* emacs
   ```
   in your command-line terminal. Do not forget to insert the correct version number at the place of the asterisk.

7. The last step is to set up the environment paths to all the executable files you just installed.

   Windows 95/98/NT:
   Edit your `autoexec.bat` file (lives in the `c:\` folder) and add these lines at the end:
   ```
   set PATH=%PATH%;c:\emacs\bin;c:\g77\bin
   set LIBRARY_PATH=c:\g77\lib
   ```
   Restart your machine.

   Windows 2k/Me/XP:
   Follow `Start > Settings > Control Panel > System`. Then select `Advanced` followed by `Environment Variables`. Find the variable `PATH`, go to its end and type `;c:\emacs\bin` followed by `;c:\g77\bin`. Then, enter a new variable (if does not exist) with name `LIBRARY_PATH` and assign it the following value: `c:\g77\lib`. Restart your machine or simply log off and log on (under Windows XP).

8. At this point, you might want to create an icon on your desktop for Emacs that you just installed. Right-click on your desktop and choose `New` and `Shortcut`. As a file name, enter
   ```
   c:\emacs\bin\runemacs.exe
   ```
   and name the icon as `Emacs`. Now, you should be in business! If it doesn't work, you made a mistake in one of the steps above. Double check and, if something goes really wrong, let me know.

If you are running MacOS X, `emacs` should have been installed automatically. Try it by typing `emacs` in your terminal window. GNU Fortran compiler is not a part of the standard distribution and has to be downloaded and installed separately. Please, check with web site `hpc.sourceforge.net` where you find under section *g77* all you need to install your `g77` compiler. The same site also provides a link to the original g77 documentation. In order to compile your source codes under MacOS X platform, it is necessary to install the Apple Developer's Toolkit from the CD that you obtained with your MacOS X (Jaguar/Panther) system.

4 Introduction to `emacs`

GNU Emacs is a free, portable, extensible text editor. That it is free means specifically that the source code is freely copyable and redistributable. That it is portable means that it runs on many machines under many different operating systems, so that you can probably count on being able to use the same editor no matter what machine you are using. That it is extensible means that you can

not only customize all aspects of its usage (from key bindings through fonts, colors, windows, mouse, and menus), but you can program Emacs to do entirely new things that its designers never thought of. Because of all this, Emacs is an extremely successful program, and does more for you than any other editor. It is particularly good for programmers. If you use a common programming language, Emacs probably provides a *mode* that makes it especially easy to edit code in that language, providing context sensitive indentation and layout.

4.1 Notation

In this document we adopt the standard Emacs notation to describe keystrokes. It is very important to become familiar with this notation, because it is routinely used in all standard Emacs documents which you can find on the Internet.

`C-x` = press the `Ctrl` key, hold it and press `x`.

`M-x` = press the `Alt` (or `Meta`) key, hold it and press `x`.

`C-M-x` = press and hold `Ctrl`, then add `Alt` (`Meta`) and then `x`. You should have three keys down at the end.

`RET` = `Enter` or `Return` key.

`SPC` = the space bar

`ESC` = the escape key

As an exercise, how would you execute the following command `C-x C-f` ? Indeed, press `Ctrl` first and keep it down while you press `x` first and then `f`. Another example may be `M-%`. Since the symbol `%` can be solely executed by pressing `Shift` and `5`, the `M-%` command is executed by first pressing `Alt`, then `Shift` and finally `5`. To learn about the most important Emacs commands which you definitely use for typing your source codes, read the following section.

4.2 Basic Emacs commands

At this point you should have your Emacs installed and your terminal shortcut ready somewhere on your desktop. Open the terminal by double-clicking on its icon, type `emacs` and execute the command by pressing `Enter`. You should now see the Emacs window with an empty *buffer* which is that part of the window which occupies the largest area. The bottom part of the window should look similarly like:

```
--:--  *scratch*      (Lisp Interaction)--L1--All--------------------------------
```

It means that you are now in the `scratch` mode in which you can practice typing, copying, deleting, searching and replacing your text. I strongly recommend you to spend some time playing with this editor prior to reading the following section. Although Emacs is an excellent programmer's editor which can help you tremendously to keep your source code clear, it requires a knowledge of certain simple operations with your text. Emacs is not like any other simple text editor and also its keyboard shortcuts differ from other editors. Do not be scared when you find that `Home` jumps to the beginning of buffer, instead of to the beginning of line. This feature can be simply changed to emulate the behavior of other editors, but this is rather advanced operation which we leave for your future exploration of Emacs.

Following is the list of the most useful commands which you will often need. Try testing each of them in the `scratch` window of Emacs to make sure that you know how they are actually executed.

`Enter` – new line
`Tab` – tabulator (the tab width is dependent on the Emacs mode)

`C-x C-f` – opens a file and shows it in the current buffer
`C-x C-s` – saves the buffer
`C-x C-w` – writes the buffer in a different file (Save As)
`C-x C-c` – quits emacs

`C-a` – jump to the beginning of the current line
`C-e` – jump to the end of the current line
`M-f` – move forward one word
`M-b` – move backward one word
`M-<` – move to the top of the buffer
`M->` – move to the bottom of the buffer
`M-x goto-char` – read a number n and move to the line number n.

`C-d` – delete the character at the cursor
`C-k` – kill (delete) the text from the position of cursor to the end of the current line
`M-d` – kill (delete) forward until the end of the next word
`M-Del` – kill (delete) backward until the beginning of a previous word

`C-s` – search forward (searching towards the end of the current buffer)
`C-r` – search backward (searching towards the top of the current buffer)
`M-%` – replace forward

`C-SPC` – mark beginning of the text for copying/moving/deleting
`C-w` – cut the text from buffer to the clipboard
`C-y` – yank (paste) the text from the clipboard at the position of the cursor

`C-_` – undo the last change
`ESC ESC ESC` – cancel the last operation (try it after `C-x C-w`)

Once you finish writing your scratch text, you can try saving it by executing `C-x C-s`. The *minibuffer* at the bottom of your screen now asks for the file name, `File to save in:`. To replace an existing file, you can always press `Tab` during writing the file name and Emacs automatically adds the rest of the name, provided that it can be uniquely identified. Pressing `Tab` once more opens a new buffer which shows you the directory structure and allows you to find the target directory manually by clicking on folders.

4.3 Emacs documentation

Emacs is a very powerful editor which contains so many features that the list given above is only a negligible part of the whole set. Apart from simple typing and rearranging your text, it allows you to also compile your source code directly from the Emacs environment, spell-check your text, find differences between an older and a more recent version of your code, write macros, etc. Although these operations cannot be thoroughly explained in the scope of this text, you may find a number of excellent resources on the Internet. A gentle tutorial to emacs can be found at

`http://www.stolaf.edu/people/humke/UNIX/emacs-tutorial.html`

To explore the more advanced functions of Emacs, consult the original Emacs manual at

`http://www.gnu.org/software/emacs/manual/`

5 Introduction to Fortran 77

5.1 F77 indentation rules

Each program written in F77 has to follow a strict column indentation rules which ensure that your source code will be correctly translated by the compiler. It means that you cannot simply write your text anywhere in the Emacs buffer you might desire, but instead the F77 standard tells you where and how a specific information has to be inserted. Here are the rules that you have to adopt when writing your F77 codes.

Column 1 of the source code designates a comment. If you place `C` (like comment), `!` (exclamation mark) or any other character in column 1, the rest of this line is ignored. If you want to comment a line which contains some code, you can place the comment *behind* the instructions (see the example below). I advise you to always comment more complex parts of your code to explain the operations which follow. It is also important to maintain your comments up-to-date after changing any critical idea originally applied in your code. Remember, you are not the only person who may work with your program. Neatly written comments help tremendously not only you but also other people to understand what the code really does.

Examples of commented lines:

```
c  Comment... It does not matter if you use small or capital 'c'
!  I prefer to use an exclamation mark to designate a comment
      .....
      do i=2,number   ! this loop calculates the factorial
         res = res*i
      enddo
      .....
```

Column 2-5 is reserved for placing a numerical *label*. The main idea is that once you place a label on a certain line, you can request an unconditional jump to this line from anywhere inside the same `program`, `function` or `subroutine` (more on these structures later).

Example of replacing the `do-enddo` loop in the example above by labeling:

```
      do 156 i=2,number   ! this loop calculates the factorial
 156  res = res*i
```

Column 6 is reserved for placing a character, usually `+`, which designates a continuation of the previous line. You will often encounter a situation in which you need to break a long line before column 73 and continue on the next line. Each continuation line must then consist in its column 6 the character `+`.

Example of the line break:

```
      write(*,'("Factorial of ",I3," is ",I5)')
     +     number, res
```

Column 7-73 is the space into which you write your instruction code that has to be translated by your Fortran compiler. Why such limited width ? Well, Fortran was born many years ago when

the final stage of coding was always punching a card which contained binary instruction code for numerically controlled devices. The standard punched card had 80 columns, numbered 1 to 80 from left to right. Each column could hold one character, encoded as some combination of punches in the 12 rows of the card; more on this at `http://www.cs.uiowa.edu/~jones/cards/codes.html`. Although many modern compilers can read your code beyond the 73th character, we will strictly cut our source code to appear within column 7 and 73. If your line would extend beyond the 73th character, you have to break it and place the symbol `+` in column 6 of each continuation line.

Fortran code should be readable not only to you, but also to anyone who might come in contact with it. To write a nice code, we will frequently indent subordinate parts of our codes, e.g. the commands inside loops, to emphasize the whole structure. Bear in mind that F77 does not impose any requirements on the number of empty lines surrounded by the instruction code and also on the number of spaces between commands. Similarly, F77 does not distinguish between upper and lower characters and so you are free to choose your own style of writing Fortran codes. In contrast to older F77 programs which are often written in capital letters, I personally prefer to use lowercase letters for everything except the names of intrinsic functions and user-defined subprograms. Nevertheless, the choice is up to you !

5.2 The first F77 program

Before we begin writing our first source code in Fortran 77, it will be very useful for you to make a special directory which will accommodate your all your Fortran codes written during this course. This can be done by typing the following command in the command-line terminal:

```
mkdir c:\work
cd c:\work
```

```
mkdir work
cd work
```

The second command moves you to the directory you just created.

Assume that you have to write a program which calculates the factorial of a particular number. To start writing your code, open your command-line terminal and run `emacs`. In the main window, press `C-x C-f`, i.e. press and hold `Ctrl` and press `x` and then `f`. In the bottom part of your `emacs` window, you should now see the following prompt:

```
Find file:  c:\work\
```

or something like `/Users/`*name*`/work/` on MacOS X, where *name* is your user name. You are now expected to enter the file name of the program you are going to write. I recommend you to always think a few seconds before you decide about the name, because a convenient file name can always help you to find the source code you seek. In our case, a good idea is to choose the name `fact.f`. The `.f` at the end should be always added, because: (i) it helps you to recognize which files contain Fortran codes, and (ii) once you use the extension `.f`, your Emacs editor automatically switches to its Fortran mode and turns on highlighting the Fortran syntax. After typing the file name, the bottom line of your screen should look as follows:

```
Find file:  c:\work\fact.f
```

or slightly differently on Mac. Now, press `Enter` and Emacs opens an empty buffer for editing your file `fact.f`. The status line at the bottom of your Emacs buffer should now read:

```
  --:--  fact.f      (Fortran)--L1--All----------------------------------------
```

which means that the file name assigned to the current buffer is `fact.f`. Because the extension of our file is `.f`, Emacs automatically recognized that we are going to write a Fortran code and switched to the Fortran mode.

Following is the program for the calculation of the factorial which we are going to type. The numbers and dashes above the code do not belong to the program and they merely serve as a "ruler" which helps you to recognize different columns. Do not type this line !

```
12--567-------------------------------------------------------------73
      program FACTORIAL

!  definition of variables
      integer i, res, number
      parameter( number=5 )

      res = 1
      do i=2,number   ! this loop calculates the factorial
         res = res*i
      enddo

      write(*,'("Factorial of ",I3," is ",I5)')
     +     number, res

      end
```

We are now at the beginning of the buffer. Press `Tab` – you should see that the cursor jumps to column 7 at which the instruction part of your code starts. Then, write `program FACTORIAL` and press `Enter`. You are done with editing the first line of your source code.

The second line is inserted by simply pressing `Enter`. I strongly recommend you to leave a blank line each time you need to distinguish between independent parts of your code. In order to write a "readable" code, you should always separate the line with identifier `program` from variables (see below), variables from the instruction part of your code and this part from the line `end`.

The third line of the code above is the *comment line*. Although comments are disregarded by your Fortran compiler, they help you to organize your source code and also help the other people to understand what your code really does. To write a comment, put an exclamation mark (`!`) in column 1 and type your comment. You should see that Emacs understood that you are writing a comment line and displayed this line with a different color.

The subsequent block always contains the *declaration of variables* which are used locally within the scope of our program. In this domain, you attach to each variable a *type* which tells your compiler whether the variable contains an integral number (`integer`), real number (`real`), logical number (`logical`), alphanumeric text (`character`), etc. Here, we use only three variables named `i`, `res` and `number`. Press `Tab` at the beginning of each line. This moves the cursor to column 7 at which F77 expects the declaration of variables. If you write more than one variable on a line, they should always be separated by a comma (`,`). The following line assigns a value to variable `number` which is declared above; more on this shortly.

The *instruction part* is the real "heart" of your code which contains the algorithm determining the factorial of a particular number `number`. Each line is typed such that you first press `Tab`. The cursor moves to the column at which the Fortran expects your input. The first line of this block, `res = 1`,

simply fills the variable `res` with number one. Press `Enter` and continue editing a new line. The `do-enddo` part is a *loop* which carries out the algebraic multiplication `1*2*3*...*number`. The first line is `do i=2,number` which means that the loop is repeated for `i`$= 2, 3, \ldots,$ `number`. Behind this header of the loop, we insert a comment which again starts with an exclamation mark (`!`). Pressing `Tab` at the beginning of the next line moves the cursor to column 10. Those three columns are added automatically to help you organize your Fortran program. Finish this line by typing `res = res*i`, which multiplies the contents of the variable `res` with the number `i` and stores the result in `res`. Finally, the tail of our loop is inserted by typing `enddo` in the subsequent line. This was a little longer part and, therefore, leave the next line blank.

Now, the part which calculates the factorial is typed. The only task left is to display the result. This is done by the instruction `write`. The asterisk (`*`) in the argument of this command means that the result will be printed by the terminal you are currently in. The second argument of `write` is the *pattern* which determines the style of output. If the command `write` were written entirely on one line, it would extend beyond column 73, which is not permitted. Therefore, we have to break the command and continue on the next line. Remember, that the symbol `+` is added in column 6 to designate that this line is a continuation of the command in the previous line. Finally, the `write` command displays `Factorial of ` followed by the value of `number`, followed by `is: ` and finished by the result of the multiplication stored in `res`.

The absolute end of your program is entered by writing `end` which is always the last instruction of your code. To save it, press `C-x C-s`, i.e. press `Ctrl`, keep holding it and press `x` and then `s`. You should see the following message in the bottom line of your Emacs editor:

```
Wrote c:\work\fact.f
```

Congratulations, your first F77 code is born. Are you eager to see the result ? Continue in the next section.

5.3 Compiling and running the code

Once you finish writing your source code, the next step is always its *compilation* which generates the executable code. To compile your code, type the following line in your terminal:

```
g77 fact.f -o fact.exe
```

This causes that `g77` compiler reads the Fortran source code `fort.f`, searches for errors in this file, translates it to the machine language and produces the executable file `fact.exe` (on Mac, you are free to avoid the extension `.exe`). Although it should not be the case here, you can sometimes encounter warnings and/or error messages produced by the compiler. If it finds errors in your source code, it also tells you what is wrong. Your task is then to go back to Emacs, find the error and fix it. Remember, the executable code is not produced unless the compilation finishes without errors.

If you correctly typed the program above, the current directory should now contain the executable file `fact.exe`. Check this by typing `dir` (on Windows) or `ls` (on Mac). Your program can now be executed by writing `fact` (on Windows) or `./fact` (on Mac) in your command-line terminal. The output of this program should look exactly as follows:

```
Factorial of   5 is   120
```

5.4 Fortran 77 documentation

A quick insight into the F77 standard can be found online at

`http://www.chem.ox.ac.uk/fortran/fortran1.html`

This site contains the most useful types of variables, loops and input/output operations you might need to use in your codes. If you seek a more complete documentation, please look at

`http://www.univ-orleans.fr/EXT/ASTEX/astex/doc/en/prof77/html/prof77.htm`

which provides almost complete description of the F77 standard. There are also many useful books on Fortran, many of them devoted already to a more recent version, Fortran 90. Once you beat F77 and decide to learn Fortran in more detail, I suggest to check with an excellent book of Ellis, Phillips, and Lahey: *Fortran 90 programming*, Addison Wesley, 1994.

6 Fundamental constructions of F77

In this section, we will go through several most important language constructions which you will mainly use when writing your first F77 codes. Every principle is explained both theoretically and practically on a simple example. At the end of each section, you will find a problem which you should try to solve. The correct solution and the necessary analysis are given for comparison.

6.1 Structure of the `program` part

We already know that each F77 code starts with the statement `program`, followed by the name of your code and ending with statement `end`. These two lines serve as an envelope of your program which is filled by the instruction code. Below is the structure of a typical F77 program:

```
program MY_PROGRAM
declaration of variables
definition of parameters
initial definition of variables
instruction part
end
```

Go back and try to identify the four parts above in our program for the calculation of the factorial.

In the following sections, we explain the meaning of each individual part in the scheme above. We will proceed from top to bottom, in the same order which you follow when writing your code. Programming is an abstract yet still very logical process in which all individual parts have their unique functions. To pass an information from one part of your code to the other, F77 uses various data structures, some of which will be explained thoroughly in subsequent lines.

6.2 Declaration of variables

A variable is represented by a symbolic name which is associated with a particular place in the memory of your computer. The value of the variable is the value currently stored in that location and this value can be changed by assigning a new value to the variable.

Prior to using a particular variable, you have to specify its *data type* or, for those of you who are more rigorous, *declare the variable*. It means that you assign a specific standard type to this variable, which in turn tells your computer to allocate a sufficient amount of space for storage of this data. This declaration of variables occurs after the statement `program`, `subroutine` or `function` (the last two subprograms will be explained later) and typically has the following form:

```
integer number, val
real result
```

In this example, `integer` and `real` are data types and `number`, `val`, `result` are symbolic names of three variables. It should be clear that the first two variables are declared to contain integral numbers, whereas the last one will store a real number. You can always assign a data type to more than one variable in one line, provided that the variables are separated by commas.

Fortran provides several standard data types which are frequently used in F77 codes. A very short compilation of the most useful ones is given below. You can see that each variable occupies a different space in the memory of your computer. For the beginning you will often use two data types, `integer` and `real`, the former of which is commonly used for counting loops and the latter contains the values used in your calculations.

symbolic name	size	description
`logical`	1 bit	stores either logical 0 (`.false.`) or logical 1 (`.true.`)
`byte`	1 byte	integral numbers from -128 to 127
`integer`	4 bytes	integral numbers from -2^{31} to $2^{31}-1$
`real`	4 bytes	real numbers approx. between $1.2 \cdot 10^{-38}$ and $3.4 \cdot 10^{38}$
`double precision`	8 bytes	real numbers approx. between $2.2 \cdot 10^{-308}$ and $1.8 \cdot 10^{308}$
`complex`	2×4 bytes	two `real` numbers, real and imaginary part
`double complex`	2×8 bytes	two `double precision` numbers, real and imaginary part
`character`	1 byte	one alphanumeric character
`character*N`	N bytes	`N` alphanumeric characters

A common task is to declare an array which can store a vector (1×N, N×1), matrix (M×N) or a higher-dimensional array (M×N×P×...). This can be done for any numerical data type by simply declaring:

```
integer vect(5)
real mat(4,10), tmat(2,5,9)
```

The variable `vect` is a vector of 5 components of type `integer` indexed from `vect(1)` to `vect(5)`, `mat` represents a matrix of `real` numbers with 4 rows and 10 columns and `tmat` is a three-dimensional array in which each element is of type `real`. The elements of the lastly-mentioned array can be accessed accordingly, e.g. `tmat(1,3,7)`.

6.3 Implicit declaration of variables

F77 allows you to assign implicitly a particular data type to all variables whose symbolic names start with a common character. This is, however, a rather advanced approach and I personally do not recommend that you use these implicit declarations. If you are curious about learning details, please feel free to consult the F77 manuals on the web. In order to switch off the implicit declarations, we always insert after the identifier `program`, `subroutine` or `function` the following line:

```
implicit none
```

This simple statement forces the compiler to strictly check whether you, as a programmer, assigned a type to each variable that you use in your code. If you omit to declare some variable(s), the compiler does not produce the executable code and exits with an error about an undeclared variable.

6.4 Definition of variables

After the declaration of variables, your computer knows how much space is to be allocated for each variable, but the actual value of this variable is not known. Although some F77 compilers implicitly assign each declared variable a zero value or an empty string (for `character` types), I strongly advise you not to count on it!

Definition of a variable is the same as assigning it a value. Assume that we have already declared `integer count`, `real result`, `logical flag`, `character*10 univ`. The definition of these variables occurs always after their declaration by assigning them their actual values:

```
count = 0
result = 5.3
flag = .false.
univ = 'upenn'
```

These statements store zero in `count`, 5.3 in `result`, logical value 0 (`.false.`) in `flag` and the string *upenn* in variable `univ`. Similar statements can also be used to define variables of other data types.

6.5 Definiton of parameters

Parameters, or constants as they are often used in other programming languages, are symbolic names which contain a value that cannot be changed during the run of your program. This is mainly convenient once you work with constants like π or e which are not implicitly defined in the F77 standard and thus have to be defined in your programs.

Before defining a parameter, you should always declare its type. These two operations occur above the instruction part of your code and might look as follows:

```
real pi, e
parameter( pi=3.2415927, e=2.7182818 )
```

Note, that it is of utmost importance to always declare the data type of each parameter prior to defining its value. If you are curious to see what happens if you do not declare the parameter type first, check with any standard F77 manual or write a short code and play with it.

6.6 Intrinsic functions

Intrinsic functions present a set of the most fundamental functions which are built into the compiler and are directly accessible in your program. These include basic mathematical functions complemented with functions for conversions between different numerical data types. Look at the table below to learn about the most useful functions which you can directly use in your codes.

function	description
`SQRT(x)`	calculates the square root of `x`
`LOG(x)`	calculates the natural logarithm of `x` (base e)
`LOG10(x)`	calculates the common log of `x` (base 10)
`EXP(x)`	raises base e to the `x` power (e^x)
`SIN(x)`	calculates the sine of `x` (`x` expressed in radians)
`COS(x)`	calculates the cosine of `x` (`x` expressed in radians)
`TAN(x)`	calculates the tangent of `x` (`x` expressed in radians)
`ASIN(x)`	calculates the inverse sine of `x` (`x` expressed in radians)
`ACOS(x)`	calculates the inverse cosine of `x` (`x` expressed in radians)
`ATAN(x)`	calculates the inverse tangent of `x` (`x` expressed in radians)
`REAL(x)`	converts the argument to a real value
`INT(x)`	converts the argument to an integer value
`NINT(x)`	rounds `x` to the nearest integer value
`ABS(x)`	calculates the absolute value of `x`
`MAX(x1,x2,...)`	returns the maximum value of `x1`, `x2`, ...

Problem: The normal (Gaussian) distribution function ϕ is defined as

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

Write an algorithm to evaluate $\phi(x)$ for values of x from -3.0 to +3.0 in steps of 0.2 and store them in array `phi`. Display the results.

Solution:

```
      real phi(50), x, pi
      parameter( pi=3.1415927 )

      x = -3.0
      do while (x .le. 3.0)
         phi(idx) = 1.0/SQRT(2*pi) * EXP(-x**2/2.0)
         write(*,'("For x = ",F4.1,", phi(x) = ",F6.4)')
     +        x, phi(idx)

         x = x + 0.2
      enddo
```

When dividing two integers or a real number and an integer, be careful to enter all integers as real numbers. For example, the expression `1/4` is not 0.25, as expected, but zero. When dividing an integer by another integer, F77 automatically rounds the result towards zero. To avoid this, you have to always enter the values in their real expressions. Hence, `1.0/4.0` correctly returns 0.25.

A slightly different case is a division of two variables of type `integer` or `byte`. Suppose that we calculate an average of `n` integral numbers whose sum is stored in variable `sum`. The average should then be calculated as `aver = REAL(sum)/REAL(n)`. Remember that `aver = sum/REAL(n)` would give the correct average if `sum` were of type `real`.

6.7 `do-enddo` loops

When writing F77 codes, you will often encounter a situation in which you need to periodically repeat a certain part of your code. This occurred also in our program `factorial`, where we had to multiply numbers from 2 to the number whose factorial is being calculated.

Generally, each `do-enddo` loop starts with statement `do par=from,to,step` by assigning the parameter `par` value `from`. The next operation is checking whether `par` $\leq$ `to` (if `step>0`) or `par` $\geq$ `to` (if `step<0`). If this condition is not true, the program jumps after the `do-enddo` body without executing the statements inside the loop. This may occur when counting the parameter with a negative step, `do i=1,5,-1`, or if the step is positive (e.g. +1) but the limits are reversed, `do i=5,1`.

At the end of each loop, the program updates the value of `par` by executing `par = par+step` and continues, provided that `par` did not exceed `to`. Once `par` exceeds the value stored in `to`, the program exits the loop and continues in interpreting the following commands. Note that if `step` is omitted, F77 automatically substitutes `step = 1`.

Problem: Write a program which calculates and prints the values of function $(1+1/n)^n$ for n from 1 to 10000 with step 100. Note that for $n \to \infty$ the function converges to e, base of the natural logarithm.

Solution:

```
      do n=1,1e4,100
         eval = (1+1.0/n)**n
         write(*,'("n = ", I5, ", e = ", F6.4)')
     +        n,eval
      enddo
```

The loop starts by assigning `n=1`. In the first execution of the loop, the program calculates the magnitude of $(1+1/n)^n$ for $n = 1$ which gives 2 and prints `n = 1, e = 2.0000`. The magnitude of `n` is then increased by 1. The second execution occurs with `n = 2`, giving `n = 2, e = 2.2500`. The loop finishes once `n` attains the value `1e4` (10000) for which the output would be `n = 10000, e = 2.7183`.

6.8 `do while-enddo` loops

In the preceding section we learned that the statement inside the `do-enddo` loop is being executed while the controlling parameter lies betwen the specified initial and final value. Remember that the decision about the execution of the statements inside the `do-enddo` construction is made *before* each individual loop.

A similar construction which is frequently used in Fortran codes is the `do while-enddo` loop. The body of the loop is executed provided that the logical expression written behind `while` is true. At the end of each cycle the program returns back to the `do while` header and again checks the validity of the logical expression. This process occurs until the condition becomes false and the program continues below `enddo`. The standard `do while-enddo` loop looks as follows:

```
do while (logical expression)
   :
   statements
   :
enddo
```

The `do while-enddo` loops are mainly used once you want to ensure that a particular loop will be executed at least once. In numerical mathematics, this frequently occurs in the algorithms for finding roots of functions where you always start with an initial guess and converge towards the configuration corresponding to zero functional value. The whole algorithm exits once you achieve a sufficient accuracy, i.e. once the functional value lies within a given ε-neighborhood of zero. To demonstrate the use of these `do while-enddo` loops, look at the following simple problem.

Problem: Write an algorithm which finds one root of function $f(x) = x^3 + 4x^2 - 1$ using a bisection method. Search the interval $0 \leq x \leq 1$ which definitely contains a root and obtain the solution with accuracy $\varepsilon \leq 10^{-6}$.

Solution:

```
eps = 1.0e-6
xleft = 0.0
xright = 1.0
fmid = eps+1.0   ! execute the first cycle

do while (ABS(fmid) .gt. eps)
   xmid = (xleft+xright)/2.0

   fleft = xleft**3 + 4*xleft**2 - 1
   fmid = xmid**3 + 4*xmid**2 - 1
   fright = xright**3 + 4*xright**2 - 1

   if (fleft*fmid .le. 0) then
      xright = xmid
   else
      xleft = xmid
   endif
enddo

write(*,'("Root is x = ",F8.6)') xmid
```

Initially, we set the requested accuracy in `eps` and specify the interval in which we are going to seek the solution. The first run of the `do while-enddo` loop calculates the position of the middle point and determines the functional values for the boundary of the interval and this middle point. The root can then lie either in the interval `(xleft,xmid)`, provided that `fleft` and `fmid` are of opposite signs, or in the interval `(xmid,xright)` if `fmid` and `fright` are of opposite signs. This new interval is then bisected and the loop repeats until the functional value corresponding to the middle point `xmid` lies in the `eps`-neighborhood of zero. When this happens, the execution of the `do while-enddo` loop is stopped and we get the result: `Root is x = 0.472834`.

The algorithm above calculates the functional values at `xleft` and `xright` at the beginning of *each* cycle, although this is not necessary in subsequent cycles of the loop. Try to understand the algorithm in more detail and improve the program to avoid these unnecessary operations. Remember that the functional value at `xmid` has to be calculated in each cycle.

6.9 `if` statements

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. This is the simplest form of the `if` statement:

```
if (logical expression)  executable statement
```

which can be used once the executable statement represents *only one* command. If more than one command is to be executed once the logical expression is true, then the following syntax should be used instead:

```
if (logical expression) then
   command1
   command2
   .....
   commandN
endif
```

The most general `if` statement comprises of 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 code is executed and the control jumps to the next statement after the `endif`. If none of the conditions is true, the program executes the statements written below the `else` identifier, in case that it exists. Note that you can always use nested `if` statements, provided that each `if` has its corresponding `endif`. Similarly, you are also free to use more than one `elseif` identifier which must, however, precede the `else` identifier, if it exists.

F77 provides the following symbols for the conditional operators which can be used in the logical expressions of the `if` statement:

operator	example	description
`.gt.`	`x.gt.y`	true if $x > y$
`.lt.`	`x.lt.y`	true if $x < y$
`.eq.`	`x.eq.y`	true if $x = y$
`.ne.`	`x.ne.y`	true if $x \neq y$
`.le.`	`x.le.y`	true if $x \leq y$
`.ge.`	`x.ge.y`	true if $x \geq y$

The logical statements can be arbitrarily combined with each other by using logical operators. Following is a table of the most important logical operators which F77 provides. Note that only `.not.` operates on a single logical expression, whereas all others operate on two expressions simultaneously.

For example, `if (x.le.y .and. y.ge.z)` is satisfied and the subsequent statement executed if and only if $(x \leq y) \wedge (y \geq z)$.

operator	example	description
`.not.`	`.not. (x.eq.0)`	true if $x \neq 0$
`.and.`	`(x.ge.0 .and. x.le.1)`	true if $0 \leq x \leq 1$
`.or.`	`(x.le.0 .or. x.ge.1)`	true if $x \in (-\infty; 0\rangle \cup \langle 1; \infty)$
`.xor.`	`x .xor. y`	true if only x or y (not both) is true (exclusive `or`)

Problem: Let us have a real number x. Write an algorithm that returns +1 if $0 < x < 1$, -1 for $x \in (-\infty; 0) \cup (1; \infty)$ and 0 otherwise.

Solution:

```
      if (x.gt.0 .and. x.lt.1) then
         ret = 1
      elseif (x.lt.0 .or. x.gt.1) then
         ret = -1
      else
         ret = 0
      endif
```

6.10 `goto` statement and its alternatives

In certain cases it may be convenient to perform an unconditional loop to another place in your code which is marked with a specific label. We already know that labels are represented by numbers written in column 2 to 5 of your Fortran code. The unconditional jump can then be requested by using a command `goto` followed by the label at which the program has to jump. Look at the following standard structure of using `goto`.

Example: Demonstration of an unconditional jump using the command `goto`.

```
      program MAIN
      integer a

      goto 11
 10   write(*,'("Here we go again")')

! lots of lines of your code

 11   write(*,'("Enter 1 to go around again:")')
      read(*,*) a
      if (a.eq.1) goto 10

      end
```

When started, this code displays `Enter 1 to go around again:` and waits for user input. After typing a number, this is assigned in `a` and the code checks whether `a` was 1. If so, it jumps to label

10, writes `Here we go again` and again asks for a number. On the other hand, if the number entered was not 1, the algorithm ends by reaching `end`.

Although you may encounter many older Fortran codes in which `goto` statements are frequently used, I recommend you to avoid using them whenever possible. As an alternative, you can always use a command `exit` which causes an immediate termination of the running loop and continuation your program after the corresponding `enddo` identifier. Another useful command is `cycle` which stops executing the current loop and goes back to the header of the loop to initiate a new cycle.

Example: Demonstration of using the commands `exit` and `cycle`.

```
      do n=1,1e4,100
         write(*,'("Executing cycle with n = ",I5)') n

         eval = (1+1.0/n)**n
         if (eval .ge. 2.718) exit

         write(*,'("  n = ", I5, ", e = ", F6.4)')
     +        n,eval
      enddo

! here we are after calling exit
```

In the code above, the `do-enddo` loop is being executed until `eval` becomes greater or equal than 2.718. Then, the loop is terminated by calling `exit` and the program continues after the `enddo` command. Now, substitute `exit` with `cycle` and think what happens. The correct answer is that whenever the calculated `eval` is greater or equal than 2.718, the result `n = ...` is not displayed, but the loop keeps going until `n` reaches 10000. The initial message `Executing cycle with n = ...` is printed in each individual cycle.

6.11 Input/output operations

An important part of any computer program is to handle input and output. In our examples so far, we used only the most common Fortran command for output, `write`. Fortran input/output processes can be quite complicated, so we will only describe some simpler cases in this tutorial.

Assume that you have a file containing data from some measurement which we want to use for further analysis. Before you can use this file you have to open it. The syntax of the `open` command is rather complicated, but for our purpose it suffices to write:

```
open(unit, FILE='file_name', STATUS='status')
```

The `unit` number is a number in the range 9-99 that denotes this file; you may choose any number but make sure you do not open two files with the same specified unit number. The `file_name` gives the name of the file which we are going to work with, e.g. `FILE='fourier.dat'`. Finally, status defines whether the file exists and has to be opened (`STATUS='old'`), it *does not exist* and will be created (`STATUS='new'`), it will be created or *replaces the old file* if exists (`STATUS='replace'`), or it is a scratch file which will be automatically deleted after closing it (`STATUS='scratch'`). Note that if you specify `STATUS='new'` and the file you are going to create already exists, the execution of the program ends up with an error. For this purpose it is convenient to use `STATUS='replace'` which first deletes the old file (if exists) and then creates and opens a new one. An opened file can be closed by the command

```
close(unit)
```

where the `unit` number identifies the file you want to close. In the case when the data are read from the terminal (input of data from user or standard output), you do not need to use `open` and `close` commands at all. The unit number for this standard input/output is substituted by symbol `*`, e.g. `write(*,*) number` displays the value of the variable `number`.

After a file has been opened with command `open`, you can access its contents using the command `read`. If the file is new and you want to store some data in it, use command `write` instead. The structure of these two commands is very similar:

```
read(unit,format)  list_of_variables
write(unit,format)  list_of_variables
```

Here, `unit` is the unit number of the file we work with, `format` specifies the form in which the data will be read/written and the `list_of_variables` provides the variables for storage the data read from the file (when reading) or the variables whose contents will be saved in file (when writing).

F77 recognizes two forms of output, namely *unformatted* and *formatted*. The former simply allows you to write data in a file or in the standard output (terminal) without specifying the `format` in the command `write`. For example, `write(*,*) number` prints the value stored in the variable `number` in your terminal window. The latter, formatted output, allows you to specify a particular format in which the data will be written/read. This is very useful to keep the outputs from your program neat for further analyses. The format can be generally built by mixing text and the so-called *descriptors*. Each descriptor defines the appearance of the output of exactly one variable given behind `read` or `write` command. Almost complete set of descriptors which you might use for defining formats in your programs is given in the table.

descriptor	meaning
Iw	output an integer in the next w character positions
F$w.d$	output a real number in the next w character positions with d decimal places
E$w.d$	output a real number in the next w character positions using an exponent format with d decimal places in the mantissa and four characters for the exponent
Aw	output a character string in the next w character positions; if w is omitted, the output will start at the next available position with no leading or trailing blanks
Lw	output `T` for true or `F` preceded by $w-1$ blanks
nX	ignore the next n character positions (n times space)
`"text"`	write the text

When using more than one descriptor, they must be separated by commas. Repetition of a certain part of the formatting pattern can be guaranteed by giving the number of repetitions, followed by a bracket containing the format to be repeated. For example: `3(F5.2,3X)` means the same as `(F5.2,3X,F5.2,3X,F5.2,3X)`. Check the examples below to get a better feel about how the format can be specified.

Example: Demonstration of the formatted output. The asterisks in the comments on the right are substitutes for blanks.

```
pi = 3.1415927
eval = 0.006272985
flag = .true.

write(*,'(F10.4)')  pi                          ! output: '****3.1416'
write(*,'("pi = ",F8.6)') pi                    ! output: 'pi = 3.141593'
write(*,'(E6.2)') eval                          ! output: '**6.27E-03'
write(*,'("This is ",L)') flag                  ! output: 'This is T'
write(*,'(2(F4.2,2X,L,3X))') pi,flag,pi,flag  ! output: '3.14**T***3.14**T***'
```

The same rules for specifying formats hold also for reading from file, e.g. `read(10,'(F5.4)') rnum`, and for reading from the standard input, e.g. `read(*,'(A80)') text`.

7 Introduction to subprograms

A fundamental approach to solving large, complex problems is to break the problem down into smaller subproblems. Then, solutions can be found for these smaller problems and organized in such a manner as to address the original problem. In a similar way, problems that require long, complex Fortran code can be broken down into program units. In addition to the main program unit beginning with identifier `program`, Fortran programs can contain many *subprograms*. Subprograms allow the programmer to organize their code in a logical, hierarchical fashion. The Fortran language provides for two types of subprograms: functions and subroutines.

7.1 Structure of `function` subprograms

A `function` subprogram is a complete, separate program that computes a single value that is returned to the main program in the function name. A function subprogram may contain any Fortran statement. Here is an example of a function which calculates an average of three numbers `a`, `b`, `c`:

```
real function AVERAGE(a,b,c)
implicit none

real a, b, c

AVERAGE = (a+b+c)/3.0

end
```

This expression says that our function has the name `AVERAGE` and returns one value of type `real` which will, of course, be the average of the three numbers `a`,`b`,`c` given as arguments. The result is *returned* from our function by statement `AVERAGE = ...`

Each function is a separate subprogram which can be called from the main `program` part, or any other function or subroutine of your code. In order to make our function `AVERAGE` accessible from a particular place in your program, you have to first declare it, similarly as we are used to declare variables:

```
real AVERAGE
```

Hence, the function `AVERAGE` can be used from the program/function/subroutine in which it was declared. Assuming that we have three numbers `num1`, `num2`, `num3`, you can calculate their average by simply calling

```
      :
      navg = AVERAGE(num1,num2,num3)
      :
```

Obviously, the data type of `navg` has to be first declared to be `real`, same as the return type of the function `AVERAGE`.

Problem: Write a function which calculates the factorial of a given number. Write a `program` part which calls this function and prints the result.

Solution:

```
      program MAIN

      integer num, res, FACT
      parameter( num=5 )

      res = FACT(num)
      write(*,'("Factorial of ",I3," is ",I5)')
     +     num, res

      end

!-----------------------------------------------

      integer function FACT(n)
      implicit none

      integer i, res, n

      res = 1
      do i=2,n
         res = res*i
      enddo

      FACT = res

      end
```

Now, it should be clear to you that the statement `res = FACT(num)` in the `MAIN` part of the code above calls the function `FACT` which calculates the factorial of `num` and returns the result in `res`. Note also, that the function `FACT` was declared by `integer FACT` before actually using it.

7.2 Structure of `subroutine` subprograms

Subroutines are similar to functions, yet differ from them in several ways. The most important one is that there is no value associated with the name of the subroutine or, in other words, subroutine

has no return statement. Subroutine is invoked using a `call` statement from anywhere else in your F77 code. Keywords `subroutine` and `end` are used to define the beginning and end of a subroutine. Compared to functions which usually have at least one argument, subroutines are often used without any arguments.

The best way to contrast the difference between a function and a subroutine is to rewrite the function for the calculation of the average of three numbers as a subroutine. This might look as follows:

```
subroutine AVERAGE(a,b,c,avg)
implicit none

real a, b, c, avg

avg = (a+b+c)/3.0

end
```

The difference between this expression and the function in the previous section is that the subroutine contains one more parameter, `avg`. This serves as a variable in which the subroutine returns the computed average of the three numbers `a`, `b` and `c`. To call this subroutine, you do not have to declare the type of `AVERAGE`, as it was needed if it were a function. Simply write:

```
:
CALL average(num1,num2,num3,navg)
:
```

The calculated average of the three numbers `num1`, `num2`, `num3` is then returned in the variable `navg`.

The most striking difference between functions and subroutines becomes obvious once you have to write a subprogram that returns more than one value. If you still doubt that you will ever need subroutines, check with the following example.

Problem: An interaction between two atoms can be described using the Lennard-Jones potential

$$\Phi(r) = 4\varepsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right] ,$$

where r is the separation of two atoms and σ, ε are adjustable constants. It was determined that the potential closely reproduces the behavior of argon if $\sigma = 3.405$ Å and $\varepsilon = 0.010323$ eV. Write a subroutine which determines the equilibrium separation of two argon atoms, r_0, for which the potential energy is at its absolute minimum. The subroutine should return both the equilibrium separation r_0 and the corresponding energy $\Phi(r_0)$.

Solution:

```
program CALCMINE
implicit none

real sigma, eps, r0, Phi0
parameter( sigma=3.405, eps=0.010323 )
```

```
      call LJ_MIN_E(sigma,eps,r0,Phi0)
      write(*,'("Equilibrium separation = ",F5.3," A")') r0
      write(*,'("Energy at equilibrium  = ",F8.6," eV")') Phi0

      end

!---------------------------------------------------------------------

      subroutine LJ_MIN_E(sigma,eps,r0,Phi0)
      implicit none

      real sigma, eps, r0, Phi0

      r0 = sigma*2**(1.0/6.0)
      Phi0 = 4*eps*((sigma/r0)**12-(sigma/r0)**6)

      end
```

The equilibrium separation r_0 corresponds to an extremum of the L-J potential, $\mathrm{d}\Phi(r)/\mathrm{d}r = 0$. This gives $r_0 = \sigma\sqrt[6]{2}$. The corresponding energy is then obtained by substituting r_0 in the expression for $\Phi(r)$. The output should look as follows:

```
Equilibrium separation = 3.822 A
Energy at equilibrium  = -.010323 eV
```

Last command which is worth mentioning here is `return` that causes an immediate end of a running subroutine and return to the program which originally called it. This is very convenient once you want to leave a subroutine in the middle and forget the remaining commands. Without using `return`, you would need to use the `goto` command for an unconditional jump to the `end` identifier of your subroutine.

8 Conclusion

The text you just read has been prepared in order to help an absolute beginner to get started with programming in Fortran 77 on Windows or MacOS X platforms. My goal was not to write a manual which would thoroughly explain a wide range of Fortran constructions, but to merely lay for you the grounds for your future study of Fortran and its application in your study and research. We omitted here many interesting parts of Fortran which can make your programming more fun. If you encounter a problem which you cannot solve based on the information you just gained, check with a good Fortran book or a devoted web site.

Remember, you will not become an experienced programmer over night, not even after two or three months of working on your assignments. Do not be afraid to use Fortran for solving more involved problems also in other courses. The best way to understand any programming language is to play with it, test different language constructions and mainly to learn from your own errors.