Chapter 2

The Baby Steps

Computers can do arithmetic, logical operation, storing and retrieving data, and communicate with users via its input/output ports. A program is needed to coordinate these elementary operations so that useful tasks can be performed. A program is then a list of directions that tells the machine what operation to perform next. In procedural languages such as FORTRAN, the directions are executed one at a time in the order they are encountered.

2.1 The notorious “Hello World”

A very simple program to get us started is listed below, the textbook lists in the F77 version and here it is in its F90 glory:

```fortran
! Anything after an exclamation mark is a comment
! This is a simple code
program hello_world       ! start and name of program, optional in F77
  print *,’Hello World!’  ! print command
  stop                    ! program terminates
end program hello_world  ! end of program , F77 accepts "end" only
```

The programs simply prints the sentence ”Hello World!” and stops. Here are some remarks about the code:

- The program is just text entered with an editor such as emacs. The text is referred to as source code.
- Any text after an exclamation mark is a comment and is not processed by the compiler. It is there for documentation purposes. The fixed
- The other lines of codes are FORTRAN statements. Some of them, such as print and stop are executable statements in that they direct the computer to do something such as print or stop the program and return control to the
operating system. Other statements, such as `program` and `end program` are non-executable and are needed by the compiler to process the source code.

- Each line not beginning with a comment sign is a statement. The end of a line denotes the end of a statement unless the line is continued with a continuation sign; in F90 that is a `&` sign at the end of the line. Comment lines cannot be continued and must start with a comment sign.

- The text shown has been entered in free-form as opposed to the fixed form required by F77. The old dialect required that the first 6 spaces be reserved for statement labels (the first 5 columns), for a continuation character (the 6th column). The line also had to end on column 72.

- The F77 standard required that all characters be capitalized. Many compiler permitted lower case characters, however, the requirement has been dropped since the F90 revision to the standard.

- The Stop statement causes the program to terminate and control is handed back to the operating system.

- The print statement causes the string constant “Hello World!” to be printed. The `*` instructs the compiler to use default formatting to print the string. Syntax rules have to be followed for the compiler to recognize the fields within a statement. The main focus here is on the semantics and meaning of the statements; and on how best to use it to structure your calculation.

### 2.2 Compiling the code

The next step is to translate the FORTRAN source code into a *machine language* native to the computer’s *processor*. The machine language calls on the processor’s instruction set to do the low level operations needed to carry out the execution of the program, like adding add two numbers. A compiler translates the FORTRAN source code into *object* files containing the machine language instructions.

In general programs must interact with the operating system to carry out the tasks (like taking input from the keyboard) and printing output on the screen. This interaction takes place via *system libraries*, essentially low level machine instructions, that must be inserted before the program is ready for execution. This insertion is called the linkage editing and is done with a program called the *loader*, Linux command `ld`. On most unix machine the compiler takes care of calling the loader automatically to link the object files and the system libraries. At this point an *executable* file is produced; its name default defaults to `a.out` on Linux machines. Notice that you cannot “read” the object files nor the executable as they are store machine language instructions.
If there are syntax errors in the source code, the compiler will trip and usually print a helpfull message indicating where the error occurred. These errors are often referred as bugs. Errors that are not syntactic errors and pass the compiler test are often revealed at run time by producing unexpected answers, most often a crash.

2.3 Running the code

On a linux machine the code is executed as follows:

```
$ ./a.out
   Hello World!
$ 
```

where the $ refers to the Linux prompt. The ./a.out is needed to indicate that the executable we want to run is located in the current directory indicated by the ./ syntax. When the command is issued the Operating System (OS) copies the file into memory and starts executing the instructions listed in it. The program is now in charge of the computer until the stop statement is executed at which point control is returned to the OS.

2.4 Wrap up

The present sections presented a "bird's eye view" of the process of creating, compiling and running a code. The details depend largely on the computer system and software used. An essential premise of high level languages is that they free us from writing the same code in machine language for every kind of processors that we need to use. Instead we decompose the work into two pieces: the application written in a high level language (in this case FORTRAN), and the compiler that takes any FORTRAN code and turns the specific application into machine language instructions. Now we need to expand on the language syntax and learn a few more tricks. The first lab is meant to fill in the details of using a Linux computer.
Chapter 3

Expressions and Assignment Statements

The “Hello World” program was used to illustrate the layout of a FORTRAN source code. Here is another example that is more numerical in character to illustrate additional concepts found in FORTRAN.

! find the hypotenuse of a right triangle
!
program hypotenuse
   implicit none

   !.Variable declaration
   real :: a           ! length of one side
   real :: b           ! length of perpendicular side
   real :: hyp         ! length of hypotenous

   a = 4.0            ! assignment of input data
   b = 3.0            ! assignment of input data

   hyp = sqrt(a**2+b**2)! calculation
   print *,hyp         ! output

   stop
end program hypotenuse

Notice the layout of the program. First is the declaration of variables and the input, then comes the calculations and finally the output. Comments are used to document the meaning of the different variables and the task of the code hypotenuse. Finally indenting the code is a good practice to delineate visually the
scope of variable and procedures. Blocks of code are likewise spatially bundled to make the code compact and to indicate close linkage.

3.1 Some intrinsic data types

The code statements involve the numbers "5.0" and "7.0" including the decimal points and the number 2 without a decimal point. These are examples of two FORTRAN intrinsic data types, namely real and integers. One has to also distinguish between constants and variables. The number 4.0 is a real constant whereas the a is a real variable that stores a value of 4.0.

3.1.1 Integers

An integer constant is a number having no fractional part. Examples include:

- 3 a number with no fractional part
- 0 an integer zero
- -32154 a large negative integer

Integers are stored exactly and their computer representation does not involve any approximation. The range of numbers that can be represented is however limited.

3.1.2 Reals

A number with a fractional part is a real. Some examples of real constants are

- 1.0 a number with a fractional part of zero
- -3.141593 an approximation to \(-\pi\)
- 6.02E+23 a large number representing \(6.02 \times 10^{23}\)
- 0.0 a real zero.

The variables 4.0 and 3.0 in the program are real constants. We use them since we expect our calculation to yield results with fractional parts. Real numbers are stored using a fraction-exponent form that will be described in more details later on. Their values are thus not stored exactly but only approximately. Round-off occurs when real numbers cannot be represented exactly.

3.2 Variables, and typing rules

The source code in hypotenuse include several variables, a, b, and hyp, which are reminiscent of those used in algebra to represent arbitrary quantities. In FORTRAN, a variable is the name of an address in memory. Variable names in fortran start with a letter, can be up to 31 characters long, and can include numerals as part of the name, such as a1 and b1.
3.3. NAMED CONSTANTS

Traditional fortran has relied on **implicit typing** rules whereby variables whose name starts with the letter, i, j, k, l, m, or n, otherwise they are real variables. This practice of implicit typing rules is actually discouraged as countless bugs are introduced by simply mistyping the name of a variable. Instead programmers are urged to use **implicit none** to override the default typing behavior and force the programmer to declare every single variable used in the code. Undeclared variables will be caught at **compile** time and flagged, instead of the user tripping on these bugs at run time. Programmers are urged to pick meaningful names for their variables to help in documenting their program. Finally notice that every new variables must be assigned a value **before** using it in a calculation.

The hypotenuse code declares 3 real variables at the beginning; two of these are input data and one is an output. The a and b variables are assigned constant real values while the hyp is assigned its value after a calculation.

A traditional variable declaration takes the form

```
intrinsic_data_type variable_name
```

For example

```
integer e
real pi
```

Fortran 90 allows additional properties to be assigned to a variable, these are **attributes**, and the declaration takes the form

```
intrinsic_data_type, attribute_list :: variable_name
```

### 3.3 Named Constants

A simple example of an attribute is to declare **named constants** using the **parameter** keyword. Parameters are variables whose value does not change throughout the calculations. For example

```
real, parameter :: gravity=9.810 ! gravitational acceleration in m/s^2
```

One reason you may want to use a named constant is that

- Avoid mistakes by not having to type long numbers multiple times, e.g. like the constant \( \pi \).
- Inform the compiler that the address cannot be written to and to crash if such an attempt is made. Also some beneficial optimization can be performed.
- If we need to change the value of the constant we need retype once only and minimizing the chances of introducing bugs.
Hence if for some reasons the units of the code need to be changed the gravity declaration would become:

```fortran
real, parameter :: gravity=32.2 ! gravitational acceleration in ft/s^2
```

### 3.4 Arithmetic Operators

The example, aside from constants and variables, uses arithmetic operators like addition + and exponentiation **. Here is a complete list of numerical operators:

- **Addition** +
- **Subtraction or negation** −
- **Multiplication** *
- **Division** /
- **Exponentiation** **

The minus sign − can be a *unary* operator when it is used to indicate a negative quantity: only one input is required as in −3.1415. All others are *binary* operators and require two inputs to carry out the calculation. The quantities involved in a binary operation should be of the same type in general except that a whole number exponent should always be an integer. Here are some side effects that we need to watch for when coding:

- **Integer division.** When one integer is divided by another the result is the integer part of the quotient, obtained by chopping off and throwing away the fractional part (not by rounding).

- **Real exponents are more costly to calculate then integer exponent.** For example x**2.0 is more expensive then x**2.

- **Occasionally the compiler will “upgrade” the variable when it sees mixed mode operations.** So 3+1.2 will be calculated as 3.0 + 1.2.

- **The order in which the operations are written makes a difference.** The order of precedence will be visited soon.

### 3.5 Function References

In addition to constants, variables and operators, the program contains a *function reference* to sqrt, which returns the positive square root of its argument. Many mathematical functions are built into FORTRAN and programmers can build their own. The syntax of the function is meant to recall the standard mathematical
3.6. EXPRESSIONS

notation in which \( f(x) \) means a function of \( x \). Some of the built-in functions are:

\[
\begin{align*}
\text{sqrt}(x) & \quad \sqrt{x} \\
\text{abs}(x) & \quad |x| \\
\text{exp}(x) & \quad e^x \\
\text{log}(x) & \quad \ln x \\
\text{mod}(n,m) & \quad \text{remainder of the division } n/m.
\end{align*}
\]

3.6 Expressions

A FORTRAN expression is made up of constants, variables, operators, function references, and parentheses supplemented with rules for computing the value of the expression. The order of precedence determines the order in which expressions are evaluated. The precedence table is listed below:

1. () expressions within parentheses are evaluated first and function references.
2. ** from right to left
3. * / from left to right
4. + - from left to right and unary negation

Quantities in parentheses are found first and have the same precedence as function evaluations. Then comes exponentiation, followed by multiplication and division on equal footing. Finally, addition and subtractions are evaluated. Where operators of equal precedence are present the expression is evaluated from left to right except for exponentiation which is evaluated from right to left. Here are some FORTRAN examples paired with their mathematical intent:

\[
\begin{align*}
2^{2^{2^3}} & \quad 2^{2^3} = 8 \\
(2^{2^2})^{2^3} & \quad (2^2)^3 = 64 \\
x/y/z/w & \quad \frac{x}{y} \frac{z}{w} \\
x*y/z*w & \quad \frac{xy}{zw}
\end{align*}
\]

Try to write expressions that are not ambiguous. When in doubt use parenthesis.

3.7 Assignment statements

The = sign in FORTRAN does not indicate an equality but is an assignment statement. It directs the compiler to store the value on the right hand of the operator into the memory location indicated by the left hand of the operator. So the FORTRAN code \( x=3.0 \) directs the compiler to store the value 3 into the address pointed to by the variable \( x \). The statement \( x=x+4.0 \) although mathematically nonsensical is perfectly valid FORTRAN code: since the right hand side expression is evaluated before assignment, the old value in \( x \) is overwritten by the new value.
3.8 Input/Output

The print statement here prints a real variable hyp. Several constants and variables of different types can be printed with a single print statement. Usually it is a good idea to confirm the input so the code could have been printed as:

```fortran
print *, 'a=', a, ' b=', b
```

The unformatted input statement read serves to read input from the keyboard. Actually in Linux it is taking input from standard input or stdin. It defaults to the keyboard unless the input is redirected from a file. So for example, the values of a and b can be obtained at run time and do not have to be “hardcoded” (so no recompiling is needed in case different values for a and b need to be put in). Here is the code:

```fortran
program hypotenuse
    implicit none
    real :: a,b,hyp

    read *, a, b
    print *, 'a=', a, ' b=', b

    hyp = sqrt(a**2+b**2) ! calculation
    print *, hyp ! output

    stop
end program hypotenuse
```

3.9 Summary

Fortran statement consists of operation acting on constants, variables, and functions. Syntax rules govern the relationship between the different fields within a statement. A number of issues have not been mentioned yet, including other FORTRAN data type, the different representation of integer and reals, and the attributes that can be assigned to variables.