Programming the computer

Programming the computer consists in producing a series of commands that the computer can understand in order to produce a desired action. Typical actions that electronic computers can perform are input and output of data, data processing, and control of interfaces or other devices.

At the most basic level, an electronic computer simply interprets voltage pulses as data or commands. From a mathematical point of view, the basic processing of information by the computer consists in the manipulation and interpretation of binary digits or bits (i.e., zeros and ones). At this level, thus, commands are represented by strings of zeros and ones.

Communicating with the computer at its most basic level takes place through the use of binary or machine language. Trying to type in these binary commands for computer programming would be overwhelmingly slow and tedious. The use of human-like language significantly facilitates the programming of computers.

The next level of computer language is referred to as assembly or assembler language. It consists of simple commands like ADD, STORE, RECALL, etc., followed by memory addresses within the computer. Although an improvement over machine language, assembly or assembler languages are still quite primitive. High-level languages such as FORTRAN 90, C++, C#, Java, Visual Basic 6.0, etc., with their human-language-like syntaxes (typically, English) facilitate the programming of the computer for most human programmers.

Each one of these high-level languages possesses its own syntaxes or language rules. Violation of the syntaxes of a specific language will result in failure of programming the computer.

Programs in high-level languages are typically typed into the computer in the form of text files, and then run through a special program referred to as an interpreter or a compiler. The interpreter or compiler translates the programs into machine language, the basic language that all computers understand.
Tools for programming
Typically, to write a program, the programmer starts by selecting or designing an algorithm, i.e., a plan for performing the action required from the computer. An algorithm can be simply a series of sequential steps that the computer must perform to produce a result. For example, if we intend to use the computer to add two numbers (a relatively simple operation, mind you), we can describe the corresponding algorithm in words as follows: *input the first number, input the second number, add the two numbers, store the resulting number into a memory location, show the number in the screen.*

Algorithm

Algorithm are logical sequence to steps or rules which solves a complex problem Mathematical, Statistical & logical it’s the frist step for programmer to solve problems

Algorithm types

1- logical

2- Numerical

Algorithm properties

1. A clear one
2. Logical sequence
3. Take all options
4. The use of flow charts

Numerical Algorithm

Algorithm are logical sequence to steps or rules which solves a Mathematical problem

Logical Algorithm

Algorithm are logical sequence to steps or rules which solves a Logical problem
Flow charts
An algorithm can be presented or described using a flowchart. A flowchart is simply a collection of geometric figures connected by arrows that illustrate the flow of the algorithmic process. The geometric figures contain information in the form of commands or mathematical operations describing the algorithm and the arrows indicate the sequence of steps in the algorithm. The following flowchart describes the algorithm written above for the addition of two numbers.

This flowchart is quite detailed for a simple operation. A more simplified version is shown below:
The basic components of a flowchart are the following geometric shapes, plus the arrows

- Start/end
- Process
- Input/output
- Decision

In the previous example, we used all but the "decision" shape. We will show examples of decision algorithms later.

**Pseudo-code**

While flowcharts are useful guides for describing an algorithm, producing a flowchart can become quite complicated, particularly if done by hand. Also, modification of a hand-made flowchart can be quite involved. Luckily, flowcharting software are now available that can simplify the process. If one doesn't want to get involved in producing a flowchart, one can use another technique known as pseudo-code.

Pseudo-code simply means writing the algorithm in brief English-like sentences that can be understood by any programmer. For example, a pseudo-code corresponding to the algorithm described in the flowchart shown earlier is presented next:

```
Start
   Input a, b
   a = b + c
   Display c
End
```

Notice the use of the algorithmic sentence `a ! b + c`, in both the flowchart and the pseudo-code, to indicate that the addition of `a` and `b` is to be stored in variable `c`.

This algorithmic sentence was translated in the Visual Basic 6.0 code as `c = a + b` because, in that high-level language, the equal sign represents an assignment operation (i.e., the value `a+b` is assigned, or stored into, `c`). Since the equal sign (`=`) represents assignment in most high-level languages, it is possible to write the sentence `n = n + 1`. This sentence, rather than representing an algebraic equality that will result in the wrong result `0 = 1`, indicates that the value
contained in variable n is to be incremented by 1, and the resulting value is to be stored into n.

**Decision structure**

A *decision structure* provides for an alternative path to the program process flow based on whether a logical statement is true or false. For example, suppose that you want to evaluate the function

\[ f(x) = \begin{cases} 
  |x-1|, & \text{if } x > -1 \\
  |x+1|, & \text{if } x \leq -1 
\end{cases} \]

The flowchart to the right indicates the process flow of a program that requests a value of x and evaluates \( y = f(x) \). The diamond contains the logical statement that needs to be checked to determine which path (T - true, or F - false) to follow. Regardless of which path is followed from the diamond, the control is returned to the *display* statement. Notice that the *input* statement and the decision statement form a sequence structure in this flowchart. As in this example, the three types of structures under consideration (sequence, decision, loop) do not appear alone, but two or three are commonly combined in many algorithms.

```plaintext
start
  input x
  if x<-1 then
    y = |x+1|
  else
    y = |x-1|
  display x,y
end
```

![Flowchart](image)
**Sequential structure**
A *sequential structure* was used in the previous section to illustrate the use of flowcharts.
Sequential structures have a single entry point and a single output point, and consist of a number of steps executed one after the other. Sequential structures can be useful in simple operations such as the addition of two numbers as illustrated earlier.

The following pseudo code illustrates a sequential structure consisting of entering a number and evaluating a function given by a single expression:

```plaintext
start
request x
calculate y = 3 \( \sin(x)/(\sin(x)+\cos(x)) \)
display x, y
end
```

**Loop structure**
A *loop structure* represents a repetition of a statement or statements a finite number of times.
For example, suppose that you want to calculate the following Summation The algorithm for the calculation is illustrated in the flowchart shown below.

```plaintext
start
input n
Sn > 0
k = 0
do while ~(k>n)
k ! k + 1
Sn= Sn + 1/k
end loop
display n, Sn
end
```
Notice that the loop structure is part of a sequence structure and that it contains a decision structure within, thus, re-emphasizing the fact that the three basic structures (sequence, decision and loops) commonly appear together in many algorithms.

Notice also that the loop structure requires an index variable, in this case $k$, to control when the process will leave the loop. The summation $S_n$ and the index variable $k$ are both initialized to zero before the control is passed to the loop structure. The first action within the loop structure is to increment the index variable $k$ by 1 ($k + 1$). Next, we check if the value of $k$ has not grown beyond that of $n$ ($k > n$?). If $k$ is still less than $n$, the control is passed to incrementing the summation ($S_n = S_n + 1/k$), and back to the first step in the loop. The process is then repeated until the condition $k > n$ is satisfied. At this point, the control is passed on to reporting the results $n$, $S_n$.

The pseudo-code corresponding to the flowchart shown above is the following:

```plaintext
start
input n
Sn = 0
k = 0
do while ~(k > n)
   k = k + 1
   Sn = Sn + 1/k
end loop
display n, Sn
end
```

Notice that instead of translating the loop structure in the flowchart with an if statement, we used the statement do while. (Do-while statements are commonly available in most high-level computer programming languages). Notice that a condition, namely ~(k > n), is attached to the do while statement in the pseudo-code. The statement is to be read as “do (the statements in the loop) while k is not larger than n”.

After the condition in the loop structure is no longer satisfied, i.e., when $k > n$, then the loop ends and the control
Programming Languages

Low-level Languages
- Assembly Languages (MV 6, SUM)
- Symbolic operation codes are used to write programs.
- Higher level but still arcane.
- Also, can be proprietary.
- Requires an assembler to translate the program into machine code.

High-level Languages
- "English-like" languages (x = 6)
- BASIC, PASCAL, FORTRAN, COBOL, C, C++, JAVA
- Requires a compiler and an assembler to translate the program.
- The compiler translates to assembly language.
- The assembler translates to machine language.