

CHAPTER 5

PROGRAMMING THE COMPUTER

5.1 An Overview of Computer Programming Languages

Basically, Human beings cannot speak or write in computer language. Likewise, computers cannot speak or write in human language therefore, an intermediate language had to be devised to allow people to communicate with the computer. These intermediate languages, known as programming languages, allow a computer programmer to direct the activities of the computer. These languages are structured around a unique set of rules that dictate exactly how a programmer should direct the computer to perform a specific task. With the powers of reasoning and logic of human beings, there is the capability to accept an instruction and understand it in many different forms. Since a computer must be programmed to respond to specific instructions, instructions cannot be given in just any form. Programming languages standardize the instruction process.

The rules of a particular language tell the programmer how the individual instructions must be structured and what sequence of words and symbols must be used to form an instruction. Generally, a computer instruction has two parts:

- operation code
- operand

The operation code tells the computer what to do such as add, subtract, multiply and divide. The operands tell the computer the data items involved in the operations. The operands in an instruction may consist of the actual data that the computer may use to perform an operation, or the storage address of data. Consider for example the instruction: $a = b + 5$. The '=' and '+' are operation codes, while 'a', 'b' and '5' are operands. The 'a' and 'b' can be storage addresses of actual data while '5' is an actual data.

Some computers use many types of operation codes in their instruction format and may provide several methods, for same thing. Other computers use fewer operation codes, but have the capacity to perform more than one operation with a single instruction. There are four basic types of instructions namely:

- (a) input-output instructions
- (b) arithmetic instructions
- (c) branching instructions and
- (d) logic instructions

An input instruction directs a computer to accept data from a specific input device and store it in a specific location in the store. An output instruction tells a computer to move a piece of data from a storage location and record it on the output medium. All basic arithmetic operations can be performed by the computer. Since arithmetic operations involve at least two numbers, an arithmetic operation must include at least two operands.

Branch instructions cause the computer to alter the sequence of execution of instruction within the program. There are two basic types of branch instructions; namely unconditional branch instruction and conditional branch instruction. An unconditional branch instruction or statement will cause the computer to branch to a statement regardless of the existing conditions. A conditional branch statement will cause the computer to branch to a statement only when certain conditions exist. Logic instructions allow the computer to change the sequence of execution of instruction, depending on conditions, built into the program by the programmer. Typical logic operations include: shift, compare and test.

5.2 Types of Programming Language

The effective utilization and control of a computer system is primarily through software. There are different types of

software that can be used to direct the computer system. System software directs the internal operations of the computer, and applications software allows the programmer to use the computer to solve user made problems. The development of programming techniques has become as important to the advancement of computer science as the developments in hardware technology. More sophisticated programming techniques and a wider variety of programming languages have enabled computers to be used in an increasing number of applications.

Programming languages, the primary means of human-computer communication, have evolved from early stages where programmers entered instructions into the computer in machine language (binary/hex) to more structured languages more similar to human language. Computer programming languages can be classified into the following categories:

- (a) Machine language
- (b) Assembly language
- (c) High level language
- (d) Very high level language

5.2.1 Machine Language

The earliest forms of computer programming were carried out by using languages that were structured according to the computer stored data, that is, in a binary number system. Programmers had to construct programs that used instructions written in binary notation, 1 and 0. Writing programs in this fashion is tedious, time-consuming and susceptible to errors.

Each instruction in a machine language program consists, as mentioned before, of two parts namely: operation code and operands. An added difficulty in machine language programming is that the operands of an instruction must tell the computer the storage address of the data to be processed. The programmer

must designate storage locations for both instructions and data as part of the programming process. Furthermore, the programmer has to know the location of every switch and register that will be used in executing the program, and must control their functions by means of instructions in the program.

A machine language program allows the programmer to take advantage of all the features and capabilities of the computer system for which it was designed. It is also capable of producing the most efficient program as far as storage requirements and operating speeds are concerned. Few (if not none) programmers today write applications programs in machine language. A machine language is computer dependent. Thus, an IBM machine language program will not run on NCR machine, REC machine or ICL machine. Machine languages are the First Generation (computer) Language (1GL).

5.2.2 Assembly (Low Level) Language

Since machine language programming proved to be a difficult and tedious task, a symbolic way of expressing machine language instructions was devised. In assembly language, the operation code is expressed as a combination of letters rather than binary numbers, sometimes called mnemonics. This allows the programmer to remember the operations codes easily than when expressed strictly as binary numbers.

The storage addresses or locations of the operands are expressed as a symbol rather than the actual numeric address. After the computer has read the program, operations software are used to establish the actual locations for each piece of data used by the program. The most popular assembly language used to be the IBM Assembly Language.

Because the computer understands and executes only machine language programs, the assembly language program must

be translated into a machine language. This is accomplished by using a system software program called an assembler. The assembler accepts an assembly language program and produces a machine language program that the computer can actually execute. The schematic diagram of the translation process of the assembly language into the machine language is shown in Fig.5.1. Although, assembly language programming offers an improvement over machine language programming, it is still an arduous task, requiring the programmer to write programs based on particular computer operation codes. An assembly language program developed and running on IBM computers would fail to run on ICL computers. Consequently, the portability of computer programs across different vendors/manufacturers of computers was not possible with assembly language. The low level languages are generally described as Second Generation Languages (2GL).

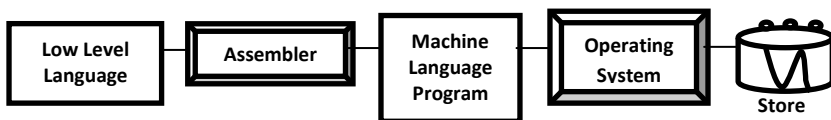


Fig. 5.1: The assembly language program translation process

5.2.3 High Level Language

The difficulty of programming and the time required to program computers in assembly languages and machine languages led to the development of high-level languages. These symbolic languages sometimes referred to as problem oriented languages reflect the type of problem being solved rather than the computer being used to solve it. Programming in machine and assembly language is machine dependent but high level languages are machine independent. In other words, a high-level language program can run on a variety of computers.

While the flexibility of high level languages is greater than that of machine and assembly languages, there are close restrictions in exactly how instructions are to be formulated and written. Only a specific set of numbers, letters, and special characters may be used to write a high level program and special rules must be observed for punctuation. High level language instructions do resemble English language statements and the mathematical symbols used in ordinary mathematics. Example of high level programming languages are FORTRAN, BASIC, COBOL, Pascal, ALGOL, ADA, PL/I, C, C++, C#, Java among others. The schematic diagram of the translation process of a high level language into the machine language is shown in Fig. 5.2. The high level languages are, generally, described as Third Generation (Computer) Language (3GL).



Fig. 5.2: The high level language program translation process

5.2.4 Very High Level Language

The very high level language generally described as the Fourth Generation (computer) Language (4GL), is an ill-defined term that refers to software intended to help computer users or computer programmers to develop their own application programs more quickly and cheaply. A 4GL, by using a menu system for example, allows users to specify what they require, rather than describe the procedures by which these requirements are met. The 4GL software does the detailed procedure by which the requirements are met, which is transparent to the users.

Programming aids or programming tools are provided to help programmers do their programming work more easily. Examples of programming tools are:

- Program development systems that help users to learn programming in a powerful high level language. Using a computer screen and keyboard directed by an interactive program, users can construct application programs.
- A program generator or application generator that assists computer users to write their own programs by expanding simple statements into program code.
- A database management system
- Debuggers which are programs that help computer users to locate errors (bugs) in the application programs they write.

A 4GL offers the user an English-like set of commands and simple control structures in which to specify general data processing or numerical operations. A program is translated into a conventional high level language such as COBOL, which is passed to a compiler. A 4GL is, therefore, a non-procedural language. The program flows are not designed by the programmer but by the fourth generation software itself. Each user request is for a result rather than a procedure to obtain the result. The conceptual diagram of the translation process of very high level language to machine language is given in Fig. 5.3.

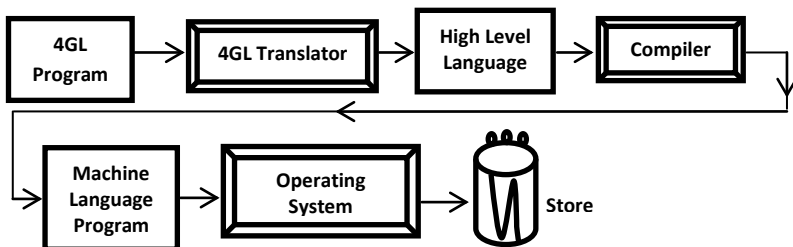


Fig. 5.3: The program translation process

The 4GL arose partly in response to the applications backlog. A great deal of programming time is spent maintaining and improving old programs rather than building new ones. Many organizations, therefore, have a backlog of applications waiting to be developed. 4GL, by stepping up the process of application design and by making it easier for end-users to build their own programs, helps to reduce the backlog.

5.3 Problem Solving with the Computer

The computer is a general-purpose machine with a remarkable ability to process information. It has many capabilities, and the user determines its specific function at any particular time. This depends on the program loaded into the computer memory being utilized by the user.

There are many types of computer programs. However, the programs designed to convert the general-purpose computer into a tool for a specific task or applications are called Application programs. These are developed by users to solve their peculiar data processing problems. Computer programming is the act of writing a program that a computer can execute to produce the desired result. A program is a series of instructions assembled to enable the computer to carry out a specified procedure. A computer program is the sequence of simple instructions into which a given problem is reduced and which is in a form the computer can understand, either directly or after translation.

5.3.1 Principles of Good Programming

It is generally accepted that a good computer program should have the following characteristics:

- **Accuracy:** The program must do what it is supposed to do correctly and meet the criteria in its specification.

- **Reliability:** The program must always do what it is supposed to do, and never crash.
- **Efficiency:** Optimal utilization of resources is essential. The program must use available storage space and other resources in such a way that system speed is not wasted.
- **Robustness:** The program should cope with invalid data and not stop without an indication of the cause/source of error.
- **Usability:** The program must be easy enough to use and be well documented.
- **Maintainability:** The program must be easy to amend, having good structuring and documentation.
- **Readability:** The code in a program should be well laid out and explained with comments.

5.3.2 Stages of Programming

The preparation of a computer program involves a set of procedure. These steps can be classified into eight major stages;

- (i) Problem definition
- (ii) Devising the method of solution
- (iii) Developing the method using suitable aids, e.g. pseudo code or flowchart.
- (iv) Writing the instructions in a programming language
- (v) Transcribing instructions into machine sensible form
- (vi) Debugging the program
- (vii) Testing the program
- (viii) Documenting all works involved in producing the program.

Problem definition

The first stage requires a good understanding of the problem. The programmer (i.e. the person writing the program)

needs to thoroughly understand what is required of a problem. A complete and precise unambiguous statement of the problem to be solved must be stated. This will entail the detailed specification which lays down the input processes and output required.

Devising the method of solution

The second stage involved is spelling out the detailed algorithm. The use of a computer to solve problems (be it scientific or business data processing problems) requires that a procedure or an algorithm be developed for the computer to follow in solving the problem.

Developing the method of solution

There are several methods for representing or developing solutions to a problem. Examples of such methods are: algorithms, flowcharts, pseudo code, and decision tables.

Writing the instructions in a programming language

After outlining the method of solving the problem, a proper understanding of the syntax of the programming language to be used is necessary in order to write the series of instructions required to get the problem solved.

Translating the instructions into machine sensible form

After the program is coded, it is converted into machine sensible form or machine language. There are some specially written programs that translate users programs (source programs) into machine language (object code). These are called translators. Compilers translate instructions that machines can execute at a go, while interpreters accept a program and execute it line-by-line. During translation, the translator carries out syntax check on the source program to detect errors that may arise from wrong use of the programming language.

Program debugging

A program might not execute successfully the first time due to the presence of a few errors (bugs). Debugging is the process of locating and correcting errors. There are three classes of errors.

- **Syntax (grammar) errors:** Caused by coding mistake (illegal use of a feature of the programming language).
- **Logic (semantic) errors:** Caused by faulty logic in the design of the program. The program will work but not as intended.
- **Execution (run-time) errors:** The program works as intended but illegal input or other circumstances at run-time makes the program stop.

There are two basic levels of debugging. The first level called desk checking or dry running is performed after the program has been coded. Its purpose is to locate and remove as many logical and clerical errors as possible. The program is then read (or loaded) into the computer and processed by a language translator. The function of the translator is to convert the program statements into the binary code of the computer called the object code. As part of the translation process, the program statements are examined to verify that they have been coded correctly, if errors are detected, the language translator generates a series of diagnostics referred to as an error message list. With this list in the hand of the programmer, the second level of debugging is reached.

The error message list helps the programmer to find the cause of errors and make the necessary corrections. At this point, the program may contain entering errors, as well as clerical errors or logic errors. The programming language manual will be very useful at this stage of program development. After corrections have been made, the program is again read into the computer and again processed by the language translator. This is repeated over and over again until the program is error-free.

Program testing

The purpose of testing is to determine whether a program consistently produces correct or expected results. A program is normally tested by executing it with a given set of input data (called test data), for which correct results are known. For effective testing of a program, the testing procedure is broken into three segments. The programmer can use any of these three alternatives to locate the bugs.

- a. The program is tested with inputs that one would normally expect for an execution of the program.
- b. Valid but slightly abnormal data is injected (used) to determine the capabilities of the program to cope with exceptions. For example, minimum and maximum values allowable for a sales amount field may be provided as input to verify that the program processed them correctly.
- c. Invalid data is inserted to test the program's error-handling routines. If the result of the testing is not adequate, then minor logic errors still abound in the program.

Other methods of testing a program for correctness include:

- **Manual walk-through:** The programmer traces the processing steps manually to find the errors, pretending to be the computer, following the execution of each statement in the program, noting whether or not the expected results are produced.
- **Use of tracing routines:** If this is available for the language's development interface, this is similar to manual walkthrough, but is carried out by the computer; hence it takes less time and is not susceptible to human error.

- **Storage dump:** This is the printout of the contents of the computer's storage locations. By examining the contents of the various locations when the program is halted, the instruction at which the program is halted can be determined. This is an important clue to finding the error that caused the halt.
- **Program documentation:** A documentation of the program should be developed at every stage of the programming cycle. The following are documentations that should be done for each program.
 - (a) **Problem Definition Step**
 - A clear statement of the problem
 - The objectives of the program (what the program is to accomplish)
 - Source of request for the program.
 - Person/official authorizing the request
 - (b) **Planning the Solution Step**
 - Flowchart, pseudo code or decision tables
 - Program narrative
 - Descriptive of input, and file formats
 - (c) **Program source coding sheet**
 - (d) **User manual to aid persons who are not familiar with the program to apply it correctly.** The manual contains a description of the program and what it is designed to achieve.
 - (e) **Operator manual to assist the computer operator to successfully run the program.** This manual contains:
 - i) Instructions about starting, running and terminating the program,
 - ii) Message that may be printed on the console or VDU (terminal) and their meanings.

iii) Setup and take down instruction for files.

Advantages of Program Documentation

- a. It provides all necessary information for anyone who comes in contact with the program.
- b. It helps the supervisor in determining the program's purpose, how long the program will be useful and future revisions that may be necessary.
- c. It simplifies program maintenance (revision or updating).
- d. It provides information as to the use of the program to those unfamiliar with it.
- e. It provides operating instructions to the computer operator.

5.4 Algorithms and Flowcharts

5.4.1 Algorithms

Before a computer can be put to any meaningful use, the user must be able to come out with or define a unit sequence of operations or activities (logically ordered) which gives an unambiguous method of solving a problem or finding out that no solution exists. Such a set of operations is known as an **ALGORITHM**.

Definition: An algorithm, named after the ninth century-scholar Abu Jafar Muhammad Ibn Musa Al-Khawarizmi, can be defined as follows:

- An algorithm is a set of rules for carrying out calculations either by hand or a machine.
- An algorithm is a finite step-by-step procedure to achieve a required result.
- An algorithm is a sequence of computational steps that transform the input into the output.

- An algorithm is a sequence of operations performed on data that have to be organized in data structures.
- An algorithm is an abstraction of a program to be executed on a physical machine (model of computation).

The most famous algorithm in history dates well before the time of the ancient Greeks: this is Euclid's algorithm for calculating the Highest Common factor (HCF) of two integers. An algorithm therefore can be characterized by the following:

- (i) A finite set or sequence of actions
- (ii) This sequence of actions has a unique initial action
- (iii) Each action in the sequence has unique successor
- (iv) The sequence terminates with either a solution or a statement that the problem is unresolved.

An algorithm can therefore be seen as a step-by-step method of solving a problem.

5.4.2 Flowchart

A flowchart is a graphical representation of the major steps in solving a task. It displays in separate boxes the essential steps of the program and shows by means of arrows the direction of information flow. The boxes, most often referred to as illustrative symbols, may represent documents, machines or actions taken during the process. A flowchart can also be said to be a graphical representation of an algorithm, that is, it is a visual picture which gives the steps of an algorithm and also the flow of control between the various steps.

5.4.3 Flowchart Symbols

Flowcharts are drawn with the help of symbols. The most commonly used flowchart symbols and their functions are shown in Fig. 5.4

5.4.4 Guidelines for Drawing Flowcharts

- Each symbol denotes a type of operation: Input, Output, Processing, Decision, Transfer or Branch or Terminal.
- A note is written inside each symbol to indicate the specific function to be performed.
- Flowcharts are read from top to bottom.
- A sequence of operations is performed until a terminal symbol designates the end of the run or branch connector transfers control.




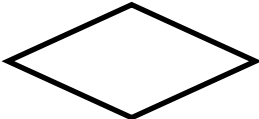



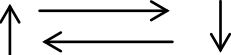
| Symbol | Function |
|---|---|
|  | Used to show the START or STOP. May show exit to a closed subordinate. |
|  | Used for arithmetic calculation of process E.g. $\text{sum} = X + Y + Z$ |
|  | Used for Input and Output instruction PRINT, READ, INPUT. |
|  | Used for decision making. Has one or more lines leaving the box. These lines are labelled with different decision Result that is, 'Yes', 'No', 'True', or 'FALSE' |
|  | Used for one or more named operations or program steps specified in a Subordinate or another set of flowchart |
|  | Used for entry to or exit from another part of flowchart. A small circle identifies one Junction point of the program. |
|  | Used for entry to or exit from a page |
|  | Used to show the direction of travel. These show operation and data flow direction |

Fig 5.5: Flowchart symbols and their functions

5.4.5 Flowcharting a Problem

The digital computer does not do any thinking and cannot make unplanned decisions. Every step of the problem has to be taken care of by the program. A problem, which can be solved by a digital computer, need not be described by an exact mathematical equation, but it does need a certain set of rules that the computer can follow. If a problem needs intuition or guessing, or is so badly defined that it is hard to put into words, the computer cannot solve it. You have to define the problem and set it up for the computer in such a way that every possible alternative is taken care of. A typical flowchart consists of special boxes, in which are written the activities or operations for solving the problem. The boxes, linked by means of arrows, show the sequence of operations. The flowchart acts as an aid to the programmer, who follows the flowchart design to write his programs.

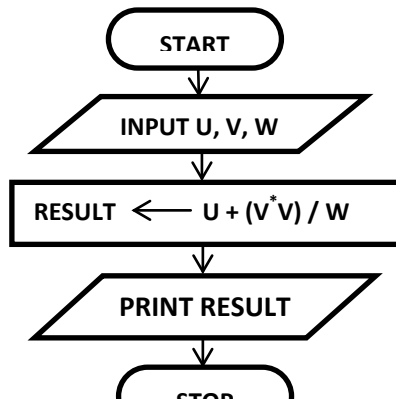
5.4.6 Examples on writing Algorithms and Flowcharting

1. Write an algorithm to read values for three variables. U, V, and W and find a value for RESULT from the formula: $RESULT = U + V^2/W$. Draw the flowchart.

Solution in Algorithm

- (i) Start
- (ii) Input values for U, V, and W
- (iii) Compute value for result, $result = U + V^2/W$
- (iv) Print value of result
- (v) Stop

Flowchart



2. Suppose you are given N numbers. Prepare the algorithm that adds up these numbers and find the average. Draw the flowchart.

Solution in Algorithm

- i. Start
- ii. Set up a Counter which counts the number of times the loop is executed. Initialize Counter to 1. Counter \leftarrow 1
- iii. Initialize sum to zero. Sum \leftarrow 0
- iv. Input value and add to sum. Input value; Sum \leftarrow Sum+value
- v. Increment the counter. counter \leftarrow counter+1
- vi. Check how many times you have added up the number, if it is not up to the required number of times, to step iv.
(if counter <N, then go to step iv)
- vii. Compute the average of the numbers. Avg \leftarrow Sum/N
- viii. Print the average.
- ix. Stop.

For Flowchart, see Fig. 5.6

3. Prepare an algorithm that prints name and weekly wages for each employee out of 10 where name, hours worked, and hourly rate are read in. Draw the flowchart.

Solution in Algorithm

- (i) Start
- (ii) Read number of workers, N
- (iii) Initialize Counter to 1 Counter \leftarrow 1
- (iv) Read name, hours and rate

- (v) Let the wage be assigned the product of hours and rate
wage ← hrs*rate
- (vi) Print name, wage
- (vii) Increment the counter by 1 Counter ← Counter+1
- (viii) Make a decision (Check how many times you have
Calculated the wages). If counter < N, then go to step iv
- (ix) Stop.

For Flowchart, see Fig. 5.7

5.4.7 Pseudocodes

A pseudocode is a program design aid that serves the function of a flowchart in expressing the detailed logic of a program. Sometimes a program flowchart might be inadequate for expressing the control flow and logic of a program. By using pseudo codes, program algorithms can be expressed as English-language statements. These statements can be used both as a guide when coding the program in a specific language and as documentation for review by others. Because there is no rigid rule for constructing pseudo codes, the logic of the program can be expressed in a manner that does not conform to any particular programming language. A series of structured words is used to express the major program functions. These structured words are the basis for writing programs using a technical term called structured programming.

Example

Construct a pseudocode for the problem in the example above.

BEGIN

STORE 0 TO SUM

STORE 1 TO COUNT

DO WHILE COUNT not greater than 10

ADD COUNT to SUM

INCREMENT COUNT by 1

ENDWILE
END

START

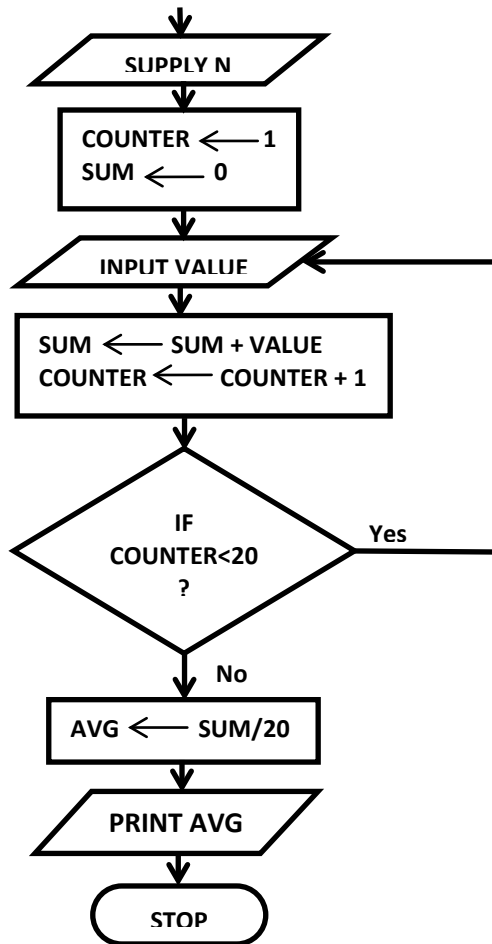


Fig. 5.6 Flowchart for Exercise 2

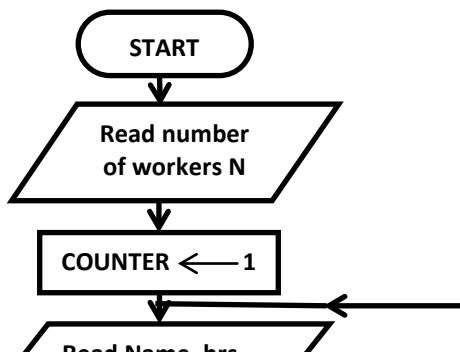


Fig. 5.7 Flowchart for Exercise 3