

# Chapter

# 4

## Computer Programming and Languages

### CHAPTER OUTLINE

- |     |                            |     |   |
|-----|----------------------------|-----|---|
| 4.1 | Introduction               | 4.6 | Generations of Programming Languages    |
| 4.2 | Problem-Solving Techniques | 4.7 | Language Translators                    |
| 4.3 | Program Control Structures | 4.8 | Features of a Good Programming Language |
| 4.4 | Programming Paradigms      |     |   |
| 4.5 | Programming Languages      |     |   |

### 4.1 INTRODUCTION

Computer has emerged as the most useful machine in recent times. It can perform wide variety of tasks like receiving data, processing it, and producing useful results. However, being a machine, the computer cannot perform on its own. A computer needs to be instructed to perform even a simple task like adding two numbers. Computers work on a set of instructions called *computer program*, which clearly specify the ways to carry out a task. An analogy of this may be thought of as the instructions given by the manager or team leader to his/her team. The team members follow those instructions and accordingly perform their duties. Similarly, a computer also takes instructions in the form of computer programs and carries out the requested task.

Now the question arises that how human beings instruct computers. We, as human beings, use natural languages, such as English, Spanish, or French to communicate. Similarly, a user communicates with the

computer in a language understood by it. Note that human beings cannot interact directly with the computer using natural languages because, thus far, we have not developed such computers that can comprehend natural languages. Rather, the instructions, provided in the form of computer programs, are developed using *computer or programming languages*. This chapter will provide some of the most prominent concepts related to computer programming and languages.

### 4.1.1 Planning the Computer Program

As discussed earlier, a program consists of a series of instructions that a computer processes to perform the required operation (Fig. 4.1). In addition, it also contains some fixed data, required to perform the instructions, and the process of defining those instructions and data. Thus, in order to design a program, a programmer must determine three basic rudiments:

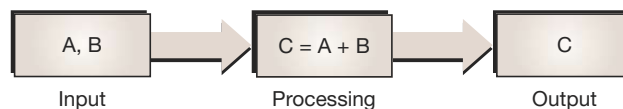


Figure 4.1 Program Performing a Task

1. The instructions to be performed.
2. The order in which those instructions are to be performed.
3. The data required to perform those instructions.

To perform a task using a program, a programmer has to consider various inputs of the program along with the process, which is required to convert the input into desired output. Suppose we want to calculate the sum of two numbers, A and B, and store the sum in C, here A and B are the inputs, addition is the process, and C is the output of the program.

#### FACT FILE

##### Development of a Program

A program is a set of instructions that instructs a computer how to perform a specific task. It is written in a high-level language that can be implemented on a number of different processors. A collection of programs can be compared to a recipe book, where each recipe can be assumed as a program. Every recipe has a list of ingredients (fixed data) and a list of instructions detailing exactly what to do with those ingredients. When you follow a recipe, you are actually executing a program.

## 4.2 PROBLEM-SOLVING TECHNIQUES

Problem-solving technique is a set of techniques and graphical tools that helps in providing logic for solving a problem. In other words, these tools are used to express the logic of the problem by specifying the correct sequence of all instructions to be carried out. Following sections discuss the various problem-solving tools, which are algorithms, flowcharts, and pseudocodes. All these tools are usually used in the design phase of the program development cycle.

### 4.2.1 Algorithms

*Algorithms* are one of the most basic tools that are used to develop the problem-solving logic. An *algorithm* is defined as a finite sequence of explicit instructions that, when provided with a set of input values, produces

an output and then terminates. In algorithm, after a finite number of steps, solution of the problem is achieved. Algorithms can have steps that repeat (iterate) or require decisions (logic and comparison) until the task is completed.

Different algorithms may accomplish the same task, with a different set of instructions, in more or less the same time, space, and efforts. For example, two different recipes for preparing tea, one “add the sugar” while “boiling the water” and the other “after boiling the water” produce the same result. However, performing an algorithm correctly does not guarantee a solution, if the algorithm is flawed or not appropriate to the context. For example, preparing the tea algorithm will fail if there were no tea leaves present; even if all the motions of preparing the tea were performed as if the tea leaves were there. We use algorithms in our daily life. For example, to determine the largest number out of three numbers A, B, and C, the following algorithm may be used.

**Step 1:** Start

**Step 2:** Read three numbers say A, B, C

**Step 3:** Find the larger number between A and B and store it in MAX\_AB

**Step 4:** Find the larger number between MAX\_AB and C and store it in MAX

**Step 5:** Display MAX

**Step 6:** Stop

The above-mentioned algorithm terminates after six steps. This explains the feature of finiteness. Every action of the algorithm is precisely defined; hence, there is no scope for ambiguity. Once the solution is properly designed, the only job left is to code that logic into a programming language. For developing an effective algorithm, flowcharts and pseudocodes are used by programmers. They are further expressed in programming languages to develop computer programs.

## 4.2.2 Flowchart

A *flowchart* is a pictorial representation of an algorithm in which the steps are drawn in the form of different shapes of boxes and the logical flow is indicated by interconnecting arrows. The boxes represent operations and the arrows represent the sequence in which the operations are implemented. The primary purpose of the flowchart is to help the programmer in understanding the logic of the program. Therefore, it is always not necessary to include all the required steps in detail. Flowcharts outline the general procedure. Since they provide an alternative, visual way of representing the information flow in a program, program developers often find them very valuable.

Flowcharts can be compared with the blueprint of a building. Just as an architect draws a blueprint before starting the construction of a building, a programmer draws a flowchart before writing a computer program. As in the case of the drawing of a blueprint, the flowchart is drawn according to defined rules and using standard flowchart symbols prescribed by American National Standard Institute (ANSI). Some standard symbols that are frequently required for flowcharts are shown in Table 4.1.





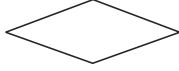



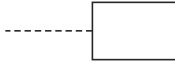
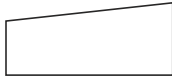


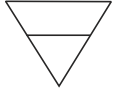
### THINGS TO REMEMBER

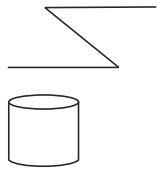
#### Properties of Algorithm

Algorithms are not computer programs, as they cannot be executed by a computer. The properties of Algorithm are:

1. There must be no ambiguity in any instruction.
2. There should not be any uncertainty about which instruction is to be executed next.
3. The algorithm should conclude after a finite number of steps. An algorithm cannot be open-ended.
4. The algorithm must be general enough to deal with any contingency.

Table 4.1 Flowchart Symbols

Symbol	Symbol Name	Description
	Flow Lines	Flow lines are used to connect symbols. These lines indicate the sequence of steps and the direction of flow of control.
	Terminal	This symbol is used to represent the beginning (start), the termination (end), or halt (pause) in the program logic.
	Input/Output	It represents information entering or leaving the system, such as customer order (input) and servicing (output).
	Processing	Process symbol is used for representing arithmetic and data movement instructions. It can represent a single step ("add two cups of flour"), or an entire sub-process ("make bread") within a larger process.
	Decision	Decision symbol denotes a decision (or branch) to be made. The program should continue along one of the two routes (IF/ELSE). This symbol has one entry and two exit paths. The path chosen depends on whether the answer to a question is yes or no.
	Connector	Connector symbol is used to join different flow lines.
	Off-page Connector	This symbol is used to indicate that the flowchart continues on the next page.
	Document	Document is used to represent a paper document produced during the flowchart process.
	Annotation	It is used to provide additional information about another flowchart symbol. The content may be in the form of descriptive comments, remarks, or explanatory notes.
	Manual Input	Manual input symbol represents input to be given by a developer/programmer.
	Manual Operation	Manual operation symbol shows that the process has to be done by a developer/programmer.
	Online Storage	This symbol represents the online data storage, such as hard disks, magnetic drums, or other storage devices.
	Offline Storage	This symbol represents the offline data storage, such as sales on OCR and data on punched cards.



Communication  
Link

Communication link symbol is used to represent data received or to be transmitted from an external system.

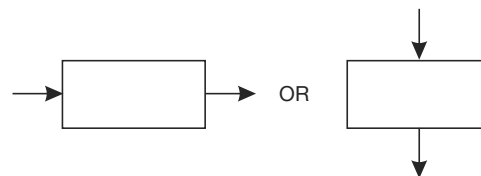


Magnetic Disk

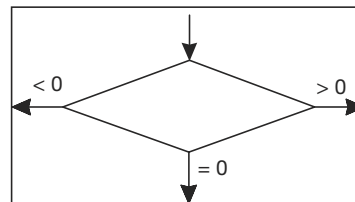
This symbol is used to represent data input or output from and to a magnetic disk.

**Guidelines for Preparing Flowcharts** The following guidelines should be used for creating a flowchart:

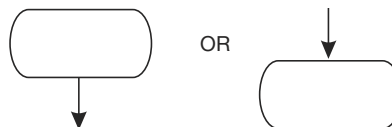
- The flowchart should be clear, neat, and easy to follow.
- The flowchart must have a logical start and finish.
- In drawing a proper flowchart, all necessary requirements should be listed in logical order.
- Only one flow line should come out from a process symbol.



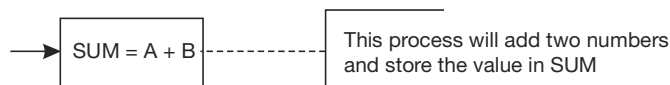
- Only one flow line should enter a decision symbol. However, two or three flow lines (one for each possible answer) may leave the decision symbol.



- Only one flow line is used with a terminal symbol.



- Within standard symbols, write briefly. If necessary, use the annotation symbol to describe data or process more clearly.



- In case of complex flowcharts, connector symbols are used to reduce the number of flow lines.
- Intersection of flow lines should be avoided to make it a more effective and better way of representing communication.
- It is useful to test the validity of the flowchart with normal/unusual test data.

**Benefits of Flowcharts** A flowchart helps to clarify how things are currently working and how they could be improved. It also assists in finding the key elements of a process by drawing clear lines between the end of one process and the start of next one. Developing a flowchart encourages communication among participants and establishes a common understanding between them about the process. Flowcharts also help in revealing redundant or misplaced steps. It also helps in establishing important areas for monitoring or data collection and to identify areas for improvement or increase in efficiency. The reasons for using flowcharts as a problem-solving tool are given below.

- **Makes Logic Clear:** The main advantage of using a flowchart to plan a task is that it provides a pictorial representation of the task, which makes the logic easier to follow. The symbols are connected in such a way that they show the movement (flow) of information through the system visibly. The steps and how each step is connected to the next can be clearly seen. Even less experienced personnel can trace the actions represented by a flowchart, that is, flowcharts are ideal for visualizing fundamental control structures employed in computer programming.
- **Communication:** Being a graphical representation of a problem-solving logic, flowcharts are better way of communicating the logic of a system to all concerned. The diagrammatical representation of logic is easier to communicate to all the interested parties as compared to actual program code as the users may not be aware of all the programming techniques and jargons.
- **Effective Analysis:** With the help of a flowchart, the problem can be analyzed in an effective way. This is because the analyzing duties of the programmers can be delegated to other persons, who may or may not know the programming techniques, but they have a broad idea about the logic. Being outsiders, they often tend to test and analyze the logic in an unbiased manner.
- **Useful in Coding:** The flowcharts act as a guide or blueprint during the analysis and program development phase. Once the flowcharts are ready, the programmers can plan the coding process effectively as they know where to begin and where to end, making sure that no steps are omitted. As a result, error-free programs are developed in high-level languages and that too at a faster rate.
- **Proper Testing and Debugging:** By nature, a flowchart helps in detecting the errors in a program, as the developers know exactly what the logic should do. Developers can test various data for a process so that the program can handle every contingency.
- **Appropriate Documentation:** Flowcharts serve as a good program documentation tool. Since normally the programs are developed for novice users, they can take the help of the program documentation to know what the program actually does and how to use the program.

**Limitations of Flowcharts** Flowchart can be used for designing the basic concept of the program in pictorial form but cannot be used for programming purposes. Some of the limitations of the flowchart are given as follows:

- **Complex:** The major disadvantage in using flowcharts is that when a program is very large, the flowcharts may continue for many pages, making them hard to follow. Flowcharts tend to get large very quickly and it is difficult to follow the represented process. It is also very laborious to draw a flowchart for a large program. You can very well imagine the nightmare when a flowchart is to be developed for a program, consisting of thousands of statements.
- **Costly:** Drawing flowcharts are viable only if the problem-solving logic is straightforward and not very lengthy. However, if flowcharts are to be drawn for a huge application, the time and cost factor of program development may get out of proportion, making it a costly affair.
- **Difficult to Modify:** Due to its symbolic nature, any change or modification to a flowchart usually requires redrawing the entire logic again, and redrawing a complex flowchart is not a simple task. It is not easy to draw thousands of flow lines and symbols along with proper spacing, especially for a large complex program.

- **No Update:** Usually programs are updated regularly. However, the corresponding update of flowcharts may not take place, especially in the case of large programs. As a result, the logic used in the flowchart may not match with the actual program's logic. This inconsistency in flowchart update defeats the main purpose of the flowcharts, that is, to give the users the basic idea about the program's logic.

### 4.2.3 Pseudocode

*Pseudocode* (pronounced *soo-doh-kohd*) is made up of two words: *pseudo* and *code*. Pseudo means imitation and code refers to instructions, written in a programming language. As the name suggests, pseudocode is not a real programming code, but it models and may even look like programming code. It is a generic way of describing an algorithm without using any specific programming language related notations. Simply put, pseudocode is an outline of a program, written in a form that can be easily converted into real programming statements. Pseudocode uses plain English statements rather than symbols to represent the processes of a computer program. It is also known as *PDL (Program Design Language)*, as it emphasizes more on the design aspect of a computer program or structured English, because usually pseudocode instructions are written in normal English, but in a structured way.

Pseudocode strikes a fine balance between the understandability and informality of a natural language like English, and the precision of a computer program code. It is somewhat halfway between English and a programming language. If an algorithm is written in English, the description may be at such a high level that it may prove difficult to analyze the algorithm and then to transform it into actual code. If instead, the algorithm is written in code, the programmer has to invest a lot of time in determining the details of an algorithm, which he may choose not to implement (since, typically, algorithms are analyzed before deciding which one to implement). Therefore, the goal of writing pseudocode is to provide a high-level description of an algorithm, which facilitates analysis and eventual coding, but at the same time suppresses many of the details that are insignificant. For example, the pseudocode given below calculates the area of a rectangle.

```
PROMPT the user to enter the height of the rectangle
PROMPT the user to enter the width of the rectangle
COMPUTE the area by multiplying the height with width
DISPLAY the area
STOP
```

Pseudocode uses some keywords to denote programming processes. Some of them are:

- **Input:** READ, OBTAIN, GET, and PROMPT
- **Output:** PRINT, DISPLAY, and SHOW
- **Compute:** COMPUTE, CALCULATE, and DETERMINE
- **Initialize:** SET and INITIALIZE
- **Add One:** INCREMENT

Since pseudocode is detailed yet readable, it can be inspected by the team of designers and programmers as a way to ensure that actual programming is likely to match design specifications. It is better to catch errors at the pseudocode stage rather than correcting them in later stages, as it would prove expensive. Once the pseudocode is accepted, it is transformed into actual program code using the vocabulary and syntax of the chosen programming language. The benefit of pseudocode is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, you can write pseudocode without even knowing what programming language you will use

for the final implementation. Often computer textbooks use pseudocode in their examples so that all programmers can understand them, even if they do not know the same programming languages.

**Pseudocode Guidelines** Writing pseudocode is not a difficult task. Even if you do not know anything about the computers or computer languages, you can still develop effective and efficient pseudocodes, if you are writing in an organized manner. Although there are no established standards for pseudocode construction, following are a few general guidelines for developing pseudocodes:

- Statements should be written in simple English (or any preferable natural language) and should be programming language independent. Remember that pseudocodes only describe the logic plan to develop a program, it is not programming.
- Steps must be understandable, and when the steps (instructions) are followed, they must produce a solution to the specified problem. If the pseudocode is difficult for a person to read or translate into code, then something is wrong with the level of detail you have chosen to use.
- Pseudocodes should be concise.
- Each instruction should be written in a separate line and each statement in pseudocode should express just one action for the computer. If the task list is properly drawn, then in most cases each task will correspond to one line of pseudocode.
- Capitalize keywords, such as READ, PRINT, and so on.
- Each set of instructions is written from top to bottom, with only one entry and one exit.
- It should allow for easy transition from design to coding in programming language.

**Benefits of Pseudocode** Programming can be a complicated process when the program requirements are complex in nature. Pseudocode provides a simple method of developing the program logic as it uses everyday language to prepare a brief set of instructions in the order in which they appear in the completed program. It allows the programmer to focus on the steps required to solve a program rather than on how to use the computer language. Some of the most significant benefits of pseudocode are as follows:

- Since it is language independent, it can be used by most programmers. It allows the developer to express the design in plain natural language.
- It is easier to develop a program from a pseudocode than with a flowchart. Programmers do not have to think about syntaxes; they simply have to concentrate on the underlying logic. The focus is on the steps to solve a problem rather than on how to use the computer language.
- Often, it is easy to translate pseudocode into a programming language, a step which can be accomplished by less-experienced programmers.
- The use of words and phrases in pseudocode, which are in line with basic computer operations, simplifies the translation from the pseudocode algorithm to a specific programming language.
- Unlike flowcharts, pseudocode is compact and does not tend to run over many pages. Its simple structure and readability make it easier to modify.

**Limitations of Pseudocode** Although pseudocode is a very simple mechanism to simplify problem-solving logic, it has its limitations. Some of the most notable limitations are:

- The main disadvantage of using pseudocode is that it does not provide visual representation of the program's logic.
- There are no accepted standards for writing pseudocodes. Programmers use their own style of writing pseudocode.
- Pseudocode cannot be compiled nor executed, and there are no real formatting or syntax rules. It is simply one step, an important one, in producing the final code (Fig. 4.2).



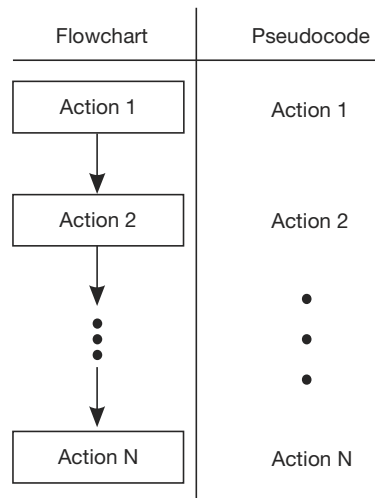


Figure 4.2 Flowchart and Pseudocode for Sequence Construct

### 4.3 PROGRAM CONTROL STRUCTURES

A program is usually not limited to a linear sequence of instructions. During its process, it may diverge, repeat code, or take decisions. For these purposes, control structures specify what has to be done to perform the program. Program statements that affect the order in which statements are executed, or that affect whether statements are executed, are called *control structures*. They affect the flow of simulation code since a control structure evaluates statements and then executes code according to the result. Essentially, there are three control structures:

1. **Sequence**, where information flows in a straight line.
2. **Selection** (decision or branched), where the decisions are made according to some predefined condition.
3. **Repetition** (looping), where the logic (sequence of steps) is repeated in a loop until the desired output is obtained.

#### 4.3.1 Sequence Control Structure

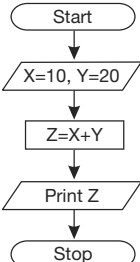
As the name implies, in a sequence structure, the instructions to be computed simply follow one another in a logical progression. This structure is denoted by writing one action after another, each action on a line by itself, and all actions aligned with the logical indent. The actions are performed in the same sequence (top to bottom) in which they are written.

Typical sequence operations consist of the process and I/O steps. The following example illustrates a simple sequence control structure that adds two numbers. Note that the actions are flowing in a logical manner, from top to bottom, as shown in Table 4.2. Not a single process branch out and no process is repeated. Each process is contributing something to the next process.

#### 4.3.2 Selection Control Structure

A selection structure allows the program to make a choice between two alternate paths, whether it is *true* or *false*. The first statement of a selection structure is a conditional statement. Once the sequence of steps in the selected path has

Table 4.2 Sequence Control Structure

Example Flowchart	Step-by-Step Instructions
	Begin the flowchart  Initialize value for memory variables X and Y  Add X and Y and store the result in variable Z  Print the value of variable Z as output  End the flowchart

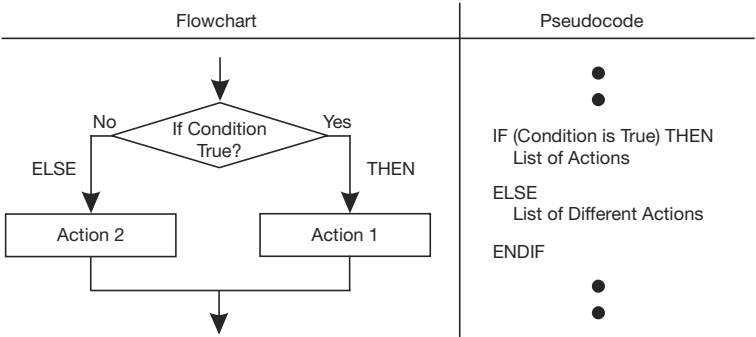
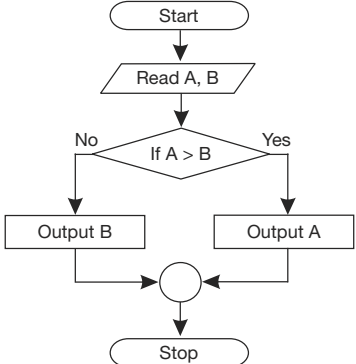


Figure 4.3 Flowchart and Pseudocode for Selection Construct

been carried out, the paths are rejoined and then the next instruction is carried out. Thus, the selection structure has only a single entry and a single exit. From Figure 4.3, it is clear that if the condition is true, Action 1 will be performed; otherwise Action 2 will be performed. After performing the list of actions, the control of the program moves on to the other actions in the process flow. The example given in Table 4.3 finds the larger of two numbers.

Table 4.3 Selection Control Structure

Example Flowchart	Step-by-Step Instructions
	Begin the flowchart  Read two numbers and store the value in memory variables A and B, respectively  Check whether the value of A is greater than the value of B  If A is greater than B then display A, otherwise display B  Connector symbols are used to join the flow lines  End the flowchart

### 4.3.3 Repetition Control Structure

Repetition or loop pattern causes an interruption in the normal sequence of processing (Fig. 4.4). It directs the system to loop back to a previous statement in the program, repeating the same sequence over and over again, usually with new data. When a sequence of statements is repeated against a condition, it is said to be in a *loop*. Using looping, the programmer avoids writing the same set of instructions again. The looping process can either be one time or multiple times until the desired output is obtained within a single program.

The following example, shown in Table 4.4, prints the first 10 natural numbers. Notice that at the beginning, the value of COUNT variable is *initialized* to zero. After that, COUNT is increased by one and the value is printed. If the value of COUNT is less than 10 (which represents the count of numbers to be displayed) then the same process is repeated. At the conclusion of each iteration, the condition is evaluated, and the loop repeats until the condition becomes true. The loop terminates when the condition becomes false.

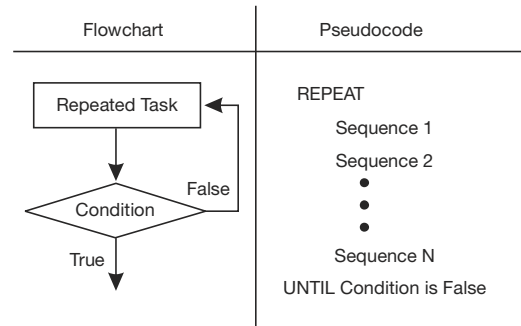


Figure 4.4 Flowchart and Pseudocode for Repetition Construct

Table 4.4 Repetition Control Structure

Example Flowchart	Step-by-Step Instructions
<pre> graph TD     Start([Start]) --&gt; Count0[Count=0]     Count0 --&gt; CountInc[Count=Count+1]     CountInc --&gt; Print[/Print Count/]     Print --&gt; IfCount{If Count&lt;10}     IfCount -- Yes --&gt; CountInc     IfCount -- No --&gt; Stop([Stop])   </pre>	<p>Begin the flowchart</p> <p>Initialize the memory variable COUNT to zero</p> <p>Increment the value of COUNT by 1</p> <p>Print the value of COUNT</p> <p>Check whether the value of COUNT is less than 10. If yes, then loop back to the third step otherwise go to the next step</p> <p>End the flowchart</p>

In all the above examples, we have used a term known as *memory variable*. Memory variable is a programming terminology used to represent the place in the memory, where the values are stored.

## 4.4 PROGRAMMING PARADIGMS

Previously, we have discussed that programming involves many hardships in terms of labour, time, and money. Usually a programmer is busy in maintaining an existing application rather than creating a new one. Hence, different types of programming paradigms have been developed in order to minimize the programming efforts. *Programming*

*paradigm* refers to how a program is written in order to solve a problem. Essentially, it is a way to think about programs and programming. It refers to the approach for developing program in order to solve a given problem.

Broadly, programming can be classified in the following three categories: *unstructured*, *structured*, and *object-oriented programming*.

4.4.1 Unstructured Programming

Unstructured style of programming refers to writing small and simple programs consisting of only one main program. All the actions, such as providing input, processing, and displaying output are done within one program only. This style of programming is generally restricted for developing a small application, but if the application becomes large then it poses real difficulties in terms of clarity of the code, modifiability, and ease of use. Although this type of programming style is not recommended, still most programmers start learning programming using this technique.

4.4.2 Structured Programming

The programs generated using unstructured approach are meant for simple and small problems. If the problem gets lengthy, this approach becomes too complex and obscure. After some time, even the programmers themselves may find it difficult to understand their own program. Hence, programming should be performed using a “structured” (organized) approach. Using structured programming, a program is broken down into small independent tasks that are small enough to be understood easily, without having to understand the whole program at once. Each task has its own functionality and performs a specific part of the actual processing. These tasks are developed independently, and each task can carry out the specified task on its own, without the help of any other task. When these tasks are completed, they are combined together to solve the problem. In this way, designers map out the large-scale structure of a program in terms of smaller operations, implement and test the smaller operations, and then tie them together into a whole program. Structured programming can be performed in two ways:

- 1. **Procedural Programming:** This programming has a single program that is divided into small pieces called *procedures* (also known as functions, routines, subroutines) as shown in Figure 4.5(a). These procedures are combined into one single location with the help of return statements. From the main or controlling procedure, a procedure call is used to invoke the required procedure. After the sequence is processed, the flow of control continues from where the call was made. The main program coordinates calls to procedures and hands over appropriate data as parameters. The data is processed by the procedures and once the whole program has finished, the resulting data is displayed.

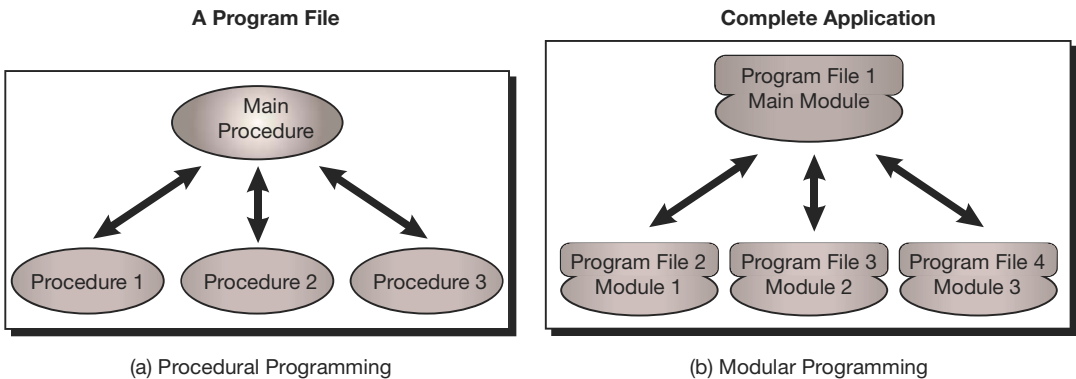


Figure 4.5 Structured Programming

2. **Modular Programming:** The programs coded with procedural paradigms usually fit into a single code file and are meant for relatively smaller programs. However, if the program gets large, then modular way of programming is recommended. In case of modular programming, large programs are broken down into a number of smaller program units known as *modules* (Fig. 4.5(b)). Each module is designed to perform a specific function. A program, therefore, is divided into several smaller parts that interact and build the whole program.

### 4.4.3 Object-Oriented Programming

Object-oriented programming (abbreviated as OOP) is a programming methodology that promotes building of independent pieces of code that interact with each other. It allows pieces of programming code to be reused and interchanged between programs. In object-oriented programming, programs are organized as cooperative collections of *objects*, each of which represents an instance of some *class*, and whose classes are all members of a hierarchy of classes united by way of *inheritance* relationships. In such programs, classes are generally viewed as static, whereas objects typically have a much more dynamic nature, which is encouraged by the existence of *polymorphism*. In OOP, the problem is first decomposed into a number of objects and then the data and functions associated with them are combined. In other words, programs are organized as a collection of objects. Thus, OOP comprises the following concepts:

- Object
- Class
- Abstraction
- Encapsulation
- Polymorphism
- Inheritance

**Object** *Objects* are the basic identifiable entities of OOP, which possess some characteristics and behaviour. For instance, an object can be a student, an employee, or a bank account. Each object has some data and functions associated with it. The data of an object specify its attributes, while the functions specify the tasks that can be performed on the object's data. An object is an instance of a class.

**Class** A *class* is a collection of similar objects. In other words, it is a template or blueprint of an object that defines its characteristics and behaviour. All objects in a given class are identical in form and behaviour but contain different data in their variables. It means that a class does not represent an object; it represents all the information a typical object should have as well as all the methods it should have.

To understand the concept of classes, let us consider an example of a class “Student” which has objects Student1, Student2, and so on, as shown in Figure 4.6. This class has some attributes (such as Name, RollNo) and

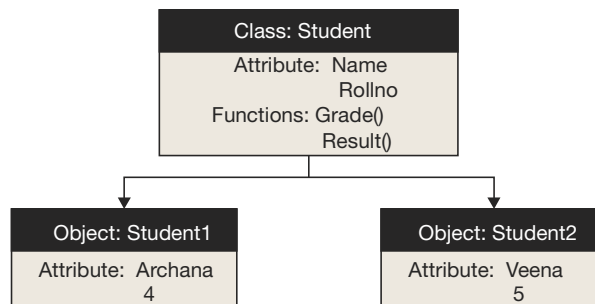


Figure 4.6 Class and Objects

functions (such as `Grade()`, `Result()`) that are common to all its objects. However, value of each attribute may be different for different objects. For example, `RollNo` of `Student1` and `Student2` will be different.

**Abstraction** *Abstraction* is the ability of a program to ignore some aspects of the information and focus on the essential elements. For example, we do not think of a car as a set of hundreds of individual parts. Rather, we view a car as an object with its own unique behaviour. This abstraction allows us to drive a car without being bothered by the complexity of the parts involved in making of the car. Hence, we can ignore the details of how the engine and braking system works and focus only on using the object, in this case, driving. Similarly, for performing a task in OOP, functions provide an interface to do so without knowing how the task has been performed and what data have been used.

**Encapsulation** Combining data and functions (that operate on that data) into a common entity called class is known as *encapsulation* (Fig. 4.7). It is a key feature of a class where data are not accessible for external or accidental modification. That is, only those functions present in the class can access the data. These functions provide the interface between the object's data and the program. This leads to what is known as *data abstraction* (or data hiding or information hiding). Hence, in OOP, abstraction is implemented through encapsulation.

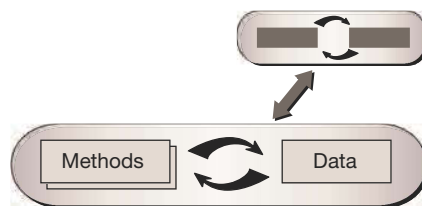


Figure 4.7 Encapsulation and Objects

**Polymorphism** The word *polymorphism* is made up of two words – *poly* which means “many” and *morphos* which means “forms.” Hence it means having more than one form. In OOP, polymorphism is the ability of a function or an object to behave differently in different situations. Let us consider an example of classes – “circle,” “triangle,” and “square” each having a function “`Area()`” to calculate their area. When a message to calculate area is sent to all of them, each reacts in a different way, as calculation of area is different for different shapes.

**Inheritance** The ability of a class to inherit properties of another class(s) is known as *inheritance*. It allows the creation of a subclass (or child class), which derives properties from another class called the base class (or parent class or super class). This leads to what is known as *code-reusability*, in which an already existing code of one class is reused to define another related class. A subclass has to define only the additional features, which are unique to it.

To understand the concept of inheritance, let us consider an example of vehicles as shown in Figure 4.8. Here the class *car* is a subclass of *Automatic Vehicle*, which is again a subclass of the class *Vehicle*. This implies that the class *car* has all the characteristics of automatic vehicle which in turn has all the properties of vehicles. However, *car* has some unique features which differentiate it from other subclasses. For example, it has four wheels and five gears, while *scooter* has two wheels and four gears.

#### 4.4.4 Characteristics of a Good Program

Every computer requires appropriate instruction set (programs) to perform the required task. The quality of the processing depends upon the given instructions. If the instructions are improper or incorrect, then it is

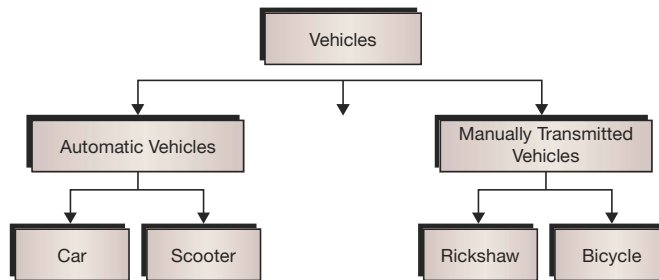


Figure 4.8 Inheritance

obvious that the result will also be superfluous. Therefore, proper and correct instructions should be provided to the computer so that it provides the correct output. Hence, a program should be developed in such a way that it ensures proper functionality of the computer. In addition, a program should be written in such a manner that it is easier to understand the underlying logic. A few important characteristics that a computer program should possess are:

- **Portability:** Portability refers to the ability of an application to run on different platform (operating systems) with or without minimal changes. Due to the rapid development in hardware and the software, nowadays platform change is a common phenomenon. Hence, if a program is developed for a particular platform then the life span of the program is severely affected.
- **Readability:** The program should be written in such a way that it makes other programmers or users follow the logic of the program without much effort. If a program is written structurally, it helps the programmers to follow their own program in a better way. Even if some computational efficiency needs to be sacrificed for better readability, it is advisable to use a more user-friendly approach, unless the application's processing is of utmost importance.
- **Efficiency:** Every program requires certain processing time and memory to process the instructions and data. As you must have realised, processing power and memory are the most precious resources of a computer; a program should be laid out in such a manner that it utilizes the least amount of memory and processing time.
- **Structural:** To develop a program, the task must be broken down into a number of subtasks. These subtasks are developed independently, and each subtask is able to perform the assigned job without the help of any other subtask. If a program is developed structurally, the program not only becomes more readable, but the testing and documentation process also gets easier.
- **Flexibility:** A program should be flexible enough to handle most of the changes without having to rewrite the entire program. Most of the programs are developed for a certain period and they require modifications from time to time. For example, in case of payroll management, as the time progresses, some employees may leave the company while some others may join. Hence, the payroll application should be flexible enough to incorporate all the changes without having to reconstruct the entire application.
- **Generality:** Apart from flexibility, the program should also be general. By generality, we mean that if a program is developed for a particular task, then it should also be used for all similar tasks of the same domain. For example, if a program is developed for a particular organization, then it should suit all the other similar organizations.
- **Documentation:** Documentation is one of the most important components of an application development. Even if a program is developed following the best programming practices, it will be rendered useless if the end user is not able to fully utilize the functionality of the application. A well-documented application is also useful for programmers because even in the absence of the author, other programmers can understand it.

## 4.5 PROGRAMMING LANGUAGES

Previously, we discussed that a computer needs to be instructed using computer programs to perform all its tasks; for this, programs are written in special computer languages. A natural language is not used to instruct the computer even though a programming language consists of a set of characters, symbols, and usage rules that allow the user to communicate with computers, just as in natural languages. The main reason behind it is that natural languages (English, Spanish) are ambiguous, vaguely structured, and have very large (and ever changing) vocabularies. Computer languages have relatively few, exactly defined, rules for composition of programs, and strictly controlled vocabularies in which unknown words must be defined before they can be used. A programming language has to follow syntax rules to create an accurate program so that the computer can yield desired results. In case of natural languages, we can understand even while using poor grammar and vocabulary. However, in case of programming language, the rules are very rigid; thus the programmer has to follow all the specified rules.

### 4.5.1 Types of Programming Languages

Computers understand only one language and that is binary language or the language of ‘0’s and ‘1’s. Binary language is also known as *machine language*. In the initial years of computer programming, all the instructions were given in binary form only. Although these programs were easily understood by the computer, it proved too difficult for a normal human being to remember all the instructions in the form of ‘0’s and ‘1’s. Therefore, the computer remained a mystery to a common person until other languages, such as assembly and high-level languages, were developed which were easier to learn and understand. These languages use commands that have some degree of similarity with English (such as “if else,” “exit”). Programming languages can be divided into three major categories:

1. **Machine Language:** It is the native language of computers. It uses only ‘0’s and ‘1’s to represent data and the instructions.
2. **Assembly Language:** It corresponds to symbolic instructions and executable machine codes and was created to use letters instead of ‘0’s and ‘1’s to run a machine.
3. **High-level Language:** These languages are written using a set of words and symbols following some rules similar to a natural language, such as English. The programs written in high-level languages are known as *source programs* and these programs are converted into machine-readable form by using compilers or interpreters.

**Note:** Together, machine and assembly language are also known as low-level languages.

## 4.6 GENERATIONS OF PROGRAMMING LANGUAGES

Since early 1950s, programming languages have evolved tremendously. This evolution has resulted in the development of hundreds of different languages. With each passing year, the languages have become user-friendly and more powerful. We can illustrate the development of all the languages in five generations.

First generation languages are machine languages, in which instructions are particular sequences of ‘0’s and ‘1’s that digital computers can understand. Second generation languages, assembly languages, allow programmers to use meaningful abbreviations for machine-specific instructions in place of the incomprehensible ‘0’s and ‘1’s form. Each instruction in an assembly language translates directly to a machine language instruction using a tool called an *assembler*. Programming became somewhat easier, but many users still wanted floating-point numbers and array indexing. Since these capabilities were not supported in hardware, high-level languages had to be



developed to support them. The next few sections discuss the five generations of languages and how they revolutionized the computer industry.

#### 4.6.1 First Generation: Machine Language

The first language was binary, also known as machine language, which was used in the earliest computers and machines. We know that computers are digital devices, which have only two states, ON and OFF (1 and 0). Hence, computers can understand only two binary codes. Therefore, every instruction and data should be written using '0's and '1's. Machine language is also known as the computer's "native" language as this system of codes is directly understood by the computer.

Instruction in machine language consists of two parts (Fig. 4.9). The first part is an opcode, which tells the computer what functions are to be performed. The second part of the instruction is the operand, which tells the computer where to find or store the data on which the desired operation is to be performed.

A binary program is a long list of instructions that are executed by the CPU. Normally, instructions are executed one after the other, but program flow may be influenced by special jump instructions that transfer execution to an instruction other than the following one. Each computer has its own set of instructions based on its architecture. Hence, machine language may differ from computer to computer.

#### THINGS TO REMEMBER

##### First Programmers

Lady Lovelace Ada Augusta (officially the first programmer) suggested binary numbers for computer storage instead of decimals. A British mathematician, Alan Mathison Turing, was the first person to recognize that programming in machine language is less time consuming. In 1952, John von Neuman proposed to have new programs loaded from a magnetic tape to read. With differences in magnetic polarities, it can mean either ON or OFF states.



Figure 4.9 Machine Language Instruction Format

**Advantages of Machine Language** Even though machine language is not a human-friendly language, it offers following advantages:

- **Translation Free:** Machine language is the only language that computers can directly execute without the need for conversion. Even an application using high-level languages, has to be converted into machine-readable form so that the computer can understand the instructions.
- **High Speed:** Since no conversion is needed, the applications developed using machine language are extremely fast. It is usually used for complex applications, such as space control system, nuclear reactors, and chemical processing.

**Disadvantages of Machine Language** There are many disadvantages in using machine language to develop programs. Some of these are as follows:

- **Machine Dependent:** Every computer type differs from the other, based on its architecture. Hence, an application developed for a particular type of computer may not run on the other type of computer. This may prove costly as well as difficult for the organizations.
- **Complex Language:** Machine language is very difficult to read and write. Since all the data and instructions must be converted to binary code, it is almost impossible to remember the instructions. A programmer must specify each operation, and the specific location for each piece of data and instruction to be stored. It means that a programmer practically needs to be a hardware expert to have proper control over the machine language.

- **Error Prone:** Since the programmer has to remember all the opcodes and the memory locations, machine language is bound to be error prone. It requires a super human effort to keep track of the logic of the problem and, therefore, results in frequent programming errors.
- **Tedious:** Machine language poses real problems while modifying and correcting a program. Sometimes the programming becomes too complex to modify and the programmer has to rewrite the entire logic again. Therefore, it is very tedious and time-consuming, and since time is a precious commodity, programming using the machine language tends to be costly.

Due to its overwhelming limitations, machine language is rarely used nowadays.

#### 4.6.2 Second Generation: Assembly Language

The complexities of machine language led to the search of another language: the assembly language, developed in the early 1950s and its main developer was IBM. However, Jack Powell, Bob Nevelen, Clement, and Michael Bradly also helped in the development of the assembly language. It was a stepping-stone for all subsequent language development. Assembly language allows the programmer to interact directly with the hardware. This language assigns a mnemonic code to each machine language instruction to make it easier to remember or write. It allows better human-readable method of writing programs as compared to writing in binary bit patterns. However, unlike other programming languages, assembly language is not a single language, but a group of languages. Each processor family (and sometimes individual processors within a processor family) has its own assembly language.

An assembly language provides a mnemonic instruction, usually three letters long, corresponding to each machine instruction. The letters are usually abbreviated indicating what the instruction does. For example, ADD is used to perform an addition operation, MUL for multiplication, and so on. Assembly languages make it easier for humans to remember how to write instructions to the computer, but an assembly language is still a representation of the computer's native instruction set. Since each type of computer uses a different native instruction set, assembly languages cannot be standardized from one machine to another, and instructions for one computer cannot be expected to work on another.

The basic unit of an assembly language program is a line of code. It allows the use of symbols and set of rules that can be used and combined to form a line of code. Each line of an assembly language program consists of four columns called *fields*. The general format of an assembly instruction is:

```
[Label] <Opcode> <Operands> [; Comment]
```

[...] brackets indicate that enclosed specification may or may not appear in a statement. If a label is specified, it is associated as a symbolic name with the machine words generated for the assembly statement. If multiple operands are used, each of them is separated by a comma. The text after semicolon (;) is just comments. Comments are not a part of actual program, but are used just for reference purposes, that is, to specify what the statement actually will do. Although comments are optional, they are included to facilitate proper documentation. For example,

Label	Opcode	Operands	Comments
BEGIN	ADD	A, B	;Add B to A

**Note:** The first character of an assembly language variable should be an alphabet. The rest of the characters may be alphabets or digits. However, the total number of characters should not exceed 8.

**Assembler** This language is nothing more than a symbolic representation of machine code, which allows symbolic designation of memory locations (Fig. 4.10). However, no matter how close assembly language is to machine code, the computer still cannot understand it. The assembly language program must be translated



Figure 4.10 Working of an Assembler

into machine code by a separate program called an *assembler*. The assembler program recognizes the character strings that make up the symbolic names of the various machine operations, and substitutes the required machine code for each instruction. At the same time, it also calculates the required address in memory for each symbolic name of a memory location, and substitutes those addresses for the names resulting in a machine language program that can run on its own at any time. In short, an assembler converts the assembly codes into binary codes and then it assembles the machine understandable code into the main memory of the computer, making it ready for execution.

The original assembly language program is also known as the *source code*, while the final machine language program is designated the *object code*. If an assembly language program needs to be changed or corrected, it is necessary to make the changes to the source code and then re-assemble it to create a new object program. The functions of an assembler are given below.

- It allows the programmer to use mnemonics while writing source code programs, which are easier to read and follow.
- It allows the variables to be represented by symbolic names, not as memory locations.
- It translates mnemonic operations codes to machine code and corresponding register addresses to system addresses.
- It checks the syntax of the assembly program and generates diagnostic messages on syntax errors.
- It assembles all the instructions in the main memory for execution.
- In case of large assembly programs, it also provides linking facility among the subroutines.
- It facilitates the generation of output on required output medium.

**Advantages of Assembly Language** The advantages of using assembly language to develop a program are as follows:

- **Easy to Understand and Use:** Assembly language uses mnemonics instead of using numerical opcodes and memory locations used in machine language. Hence, the programs written in assembly language are much more easy to understand and use as compared to its machine language counterpart. Being a more user-friendly language as compared to machine language, assembly programs are easier to modify.
- **Less Error Prone:** Since mnemonic codes and symbolic addresses are used, the programmer does not have to keep track of the storage locations of the information and instructions. Hence, there are fewer errors while writing an assembly language program. Even in case of errors, assembly programs provide better facility to locate and correct them as compared to machine language programs. Moreover, assemblers also provide various mechanisms to locate the errors. For example, in case of adding two variables, such as ADD A, B, if the variables (A and B) are not defined in the program, the assembler will give an error indicating the same so that the programmer can easily correct the mistake.
- **Faster:** Assembly programs can run much faster and use less memory and other resources than a similar program written in a high-level language. Speed increment of 2 to 20 times faster is common, and occasionally, an increase of hundreds of times faster is also possible.

- **More Control on Hardware:** Assembly language also gives direct access to key machine features essential for implementing certain kinds of low-level routines, such as an operating system kernel or micro-kernel, device drivers, and machine control.

**Disadvantages of Assembly Language** The disadvantages in using assembly language to develop a program are:

- **Machine Dependent:** Different computer architectures have their own machine and assembly languages, which means that programs written in these languages are not portable to other, incompatible systems. This makes it a low-level language. If an assembly program is to be shifted to a different type of computer, it has to be modified to suit the new environment.
- **Harder to Learn:** The source code for an assembly language is cryptic and in a very low machine-specific form. Being a machine-dependent language, every type of computer architecture requires a different assembly language, making it hard for a programmer to remember and understand every dialect of assembly. More skilled and highly trained programmers, who know all about the logical structure of the computer only, can create applications using assembly language.
- **Slow Development Time:** Even with highly skilled programmers, assembly generated applications are slower to develop as compared to high-level language based applications. In case of assembly language, the development time can be 10 to 100 times as compared to high-level language generated application.
- **Less Efficient:** A program written in assembly language is less efficient than machine language because every assembly instruction has to be converted into machine language. Therefore, the execution of assembly language program takes more time than machine language program. Moreover, before executing an assembly program, the assembler has to be loaded in the computer's memory for translation and occupies a sizeable memory.
- **No Standardization:** Assembly languages cannot be standardized because each type of computer has a different instruction set and, therefore, a different assembly language.
- **No Support for Modern Software Engineering Technology:** Assembly languages provide no inherent support for software engineering technology. They work with just machine-level specifics, not with abstractions. Assembly language does not provide inherent support for safety-critical systems. It provides very little opportunity for reuse and there is no object-oriented programming support. There is also no specific support for distributed systems. The tools available for working with assembly languages are typically very low-level tools.

#### 4.6.3 Third Generation: High-level Language

During 1960s, computers started to gain popularity and it became necessary to develop languages that were more like natural languages, such as English so that a common user could use the computer efficiently. Since assembly language required deep knowledge of computer architecture, it demanded programming as well as hardware skills to use computers. Due to computer's widespread usage, early 1960s saw the emergence of the third generation programming languages (3GL). Languages, such as COBOL, FORTRAN, BASIC, and C are examples of 3GLs and are considered high-level languages.

High-level languages are similar to English language. Programs written using these languages can be machine independent. A single high-level statement can substitute several instructions in machine or assembly language. Unlike assembly and machine programs, high-level programs may be used with different types of computers with little or no modification, thus reducing the re-programming time.

In high-level language, programs are written in a sequence of statements to solve a problem. For example, the following BASIC code snippet will calculate the sum of two numbers:

```
LET X = 10
LET Y = 20
LET SUM = X + Y
PRINT SUM
```

The first two statements store 10 in variable X (memory location name) and 20 in variable Y, respectively. The third statement again creates a variable named SUM, which will store the summation of X and Y value. Finally, the output is printed, that is, the value stored in SUM is printed on the screen.

**Advantages of High-Level Languages** High-level languages (HLL) are useful in developing complex software, as they support complex data structures. It increases the programmer's productivity (the number of lines of code generated per hour). Unlike assembly language, the programmer does not need to learn the instruction set of each computer being worked with. The various advantages of using HLLs are discussed below.

- **Readability:** Since HLLs are closer to natural languages, they are easier to learn and understand. In addition, a programmer does not need to be aware of computer architecture; even a common man can use it without much difficulty. This is the main reason of HLL's popularity.
- **Machine Independent:** High-level languages are machine independent in the sense that a program created using HLL can be used on different platforms with very little or no change at all.
- **Easy Debugging:** HLLs include the support for ideas of abstraction so that programmers can concentrate on finding the solution to the problem rapidly, rather than on low-level details of data representation, which results in fewer errors. Moreover, the compilers and interpreters are designed in such a way that they detect and point out the errors instantaneously.
- **Easier to Maintain:** As compared to low-level languages, the programs written in HLL are easy to modify and maintain because HLL programs are easier to understand.
- **Low Development Cost:** HLLs permit faster development of programs. Although a high-level program may not be as efficient as an equivalent low-level program, the savings in programmer's time generally outweighs the inefficiencies of the application. This is because the cost of writing a program is nearly constant for each line of code, regardless of the language. Thus, a high-level language, where each line of code translates to 10 machine instructions, costs only a fraction as compared to a program developed in a low-level language.
- **Easy Documentation:** Since the statements written in HLL are similar to natural languages, they are easier to understand as compared to low-level languages.

**Disadvantages of High-Level Languages** The main disadvantages of this language are:

- **Poor Control on Hardware:** HLLs are developed to ease the pressure on programmers so that they do not have to know the intricacies of hardware. As a result, sometimes the applications written in HLLs cannot completely harness the total power available at hardware level.
- **Less Efficient:** The HLL applications are less efficient as far as computation time is concerned. This is because, unlike low-level languages, HLLs must be created and sent through another processing program known as a *compiler*. This process of translation increases the execution time of an application. Programs written in HLLs take more time to execute, and require more memory space. Hence, critical applications are generally written in low-level languages.

#### 4.6.4 Fourth Generation: 4GL

Fourth generation languages (4GLs) have simple, English-like syntax rules, commonly used to access databases. The third generation programming languages are considered as procedural languages because the programmer

must list each step and must use logical control structures to indicate the order in which instructions are to be executed. 4GLs, on the other hand, are non-procedural languages. The non-procedural method is simply to state the needed output instead of specifying each step one after another to perform a task. In other words, the computer is instructed *what* it must do rather than *how* a computer must perform a task.

The non-procedural method is easier to write, but has less control over how each task is actually performed. When using non-procedural languages, the methods used and the order in which each task is carried out is left to the language itself; the user does not have any control over it. In addition, 4GLs sacrifice computer efficiency in order to make programs easier to write. Hence, they require more computer power and processing time. However, with the increase in power and speed of hardware and with diminishing costs, the uses of 4GLs have spread.

Fourth generation languages have a minimum number of syntax rules. Hence, common people can also use such languages to write application programs. This saves time and allows professional programmers for more complex tasks. The 4GLs are divided into three categories:

1. **Query Languages:** They allow the user to retrieve information from databases by following simple syntax rules. For example, the database may be requested to locate details of all employees drawing a salary of more than \$10000. Structured Query Language (SQL) and IBM's Query-By-Example (QBE) are examples of query languages.
2. **Report Generators:** They produce customized reports using data stored in a database. The user specifies the data to be in the report, the report's format, and whether any subtotals and totals are needed. Often report specifications are selected from pull-down menus, making report generators very easy to use. Examples of report generators are Easytrieve Plus by Pansophic and R&R Relational Report Writer by Concentric Data Systems.
3. **Application Generators:** With application generators, the user writes programs to allow data to be entered into the database. The program prompts the user to enter the needed data. Cincom System's MANTIS and ADS by Cullinet are examples of application generators.

**Advantages of 4GLs** The main advantage of 4GLs is that a user can create an application in a much shorter time for the development and debugging than with other programming languages. The programmer is only interested in what has to be done and that too at a very high level. Being non-procedural in nature, it does not require the programmers to provide the logic to perform a task. As a result, lot of programming effort is saved. Use of procedural templates and data dictionaries allow automatic type checking (for the programmer and for user input) and this results in fewer errors. Using application generators, the routine tasks are automated.

**Disadvantages of 4GLs** Since programs written in a 4GL are quite lengthy, they need more disk space and a large memory capacity as compared to 3GLs. These languages are inflexible also because the programmers' control over language and resources is limited as compared to other languages. These languages cannot directly utilize the computer power available at hardware level as compared to other levels of languages.

#### 4.6.5 Fifth Generation: Very High-level Languages

Fifth generation languages are just the conceptual view of what might be the future of programming languages. These languages will be able to process natural languages. The computers would be able to accept, interpret, and execute instructions in the native or natural language of the end users. The users will be free from learning any programming language to communicate with the computers. The programmers may simply type the instruction or simply tell the computer by way of microphones what it needs to do. Since these languages are still in their infancy, only a few are currently commercially available. They are closely linked to artificial intelligence and expert systems.



## 4.7 LANGUAGE TRANSLATORS

Since computers understand only machine language, it is necessary to convert the HLL programs into machine language codes. This is achieved by using language translators or language processors, generally known as compilers, interpreters, or other routines that accept statements in one language and produce equivalent statements in another language.

**Compiler** A compiler is a kind of translator that translates a program into another program, known as *target language* (Fig. 4.11). Usually, the term compiler is used for language translator of high-level language into machine language. The compiler replaces single high-level statement with a series of machine language instruction. A compiler usually resides on a disk or other storage media. When a program is to be compiled, its compiler is loaded into main memory. The compiler stores the entire high-level program, scans it, and translates the whole program into an equivalent machine language program. During the translation process, the compiler reads the source program and checks the syntax (grammatical) errors (Fig. 4.12). If there is any error, the compiler generates an error message, which is usually displayed on the screen. In case of errors, the compiler will not create the object code until all the errors are rectified.

Once the program has been compiled, the resulting machine code is saved in an executable file, which can be run on its own at any time. To be precise, once the executable file is generated, there is no need for the actual

### THINGS TO REMEMBER

#### Compilers

Compiler is a program that translates sources code written in a particular programming language into computer-readable machine code that can be directly loaded and executed. For each HLL, a separate compiler is required. For example, a compiler for C language cannot translate a program written in FORTRAN. Hence, to host computer must have the compilers of both languages.

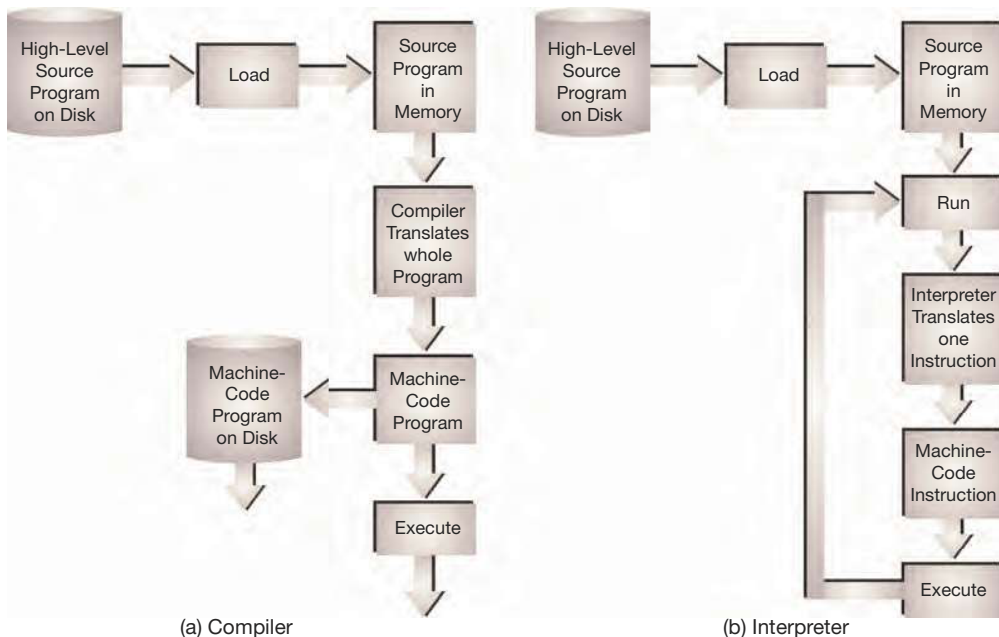


Figure 4.11 Working of Compiler and Interpreter

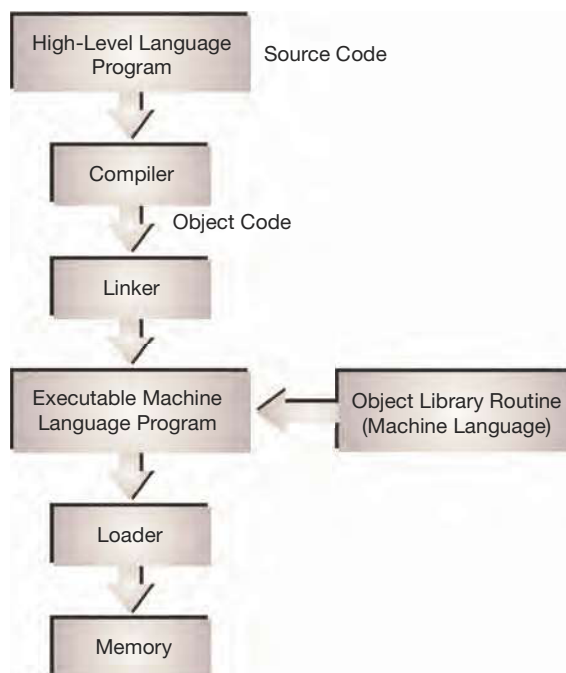


Figure 4.12 Program Translation Hierarchy

source code file. Anyway, it is worthwhile to keep the source file(s) because if the source code is modified, it is necessary to recompile the program again to regenerate the executable file containing amendments.

**Interpreter** Unlike compilers, an interpreter translates a statement in a program and executes the statement immediately, before translating the next source language statement. When an error is encountered in the program, the execution of the program is halted and an error message is displayed. Similar to compilers, every interpreted language, such as BASIC and LISP has its own interpreters.

**Linker** An application usually consists of hundreds or thousands of lines of codes. The codes are divided into logical groups and stored in different modules so that the debugging and maintenance of the codes become easier. Hence, for an application, it is always advisable to adhere to structural (modular) programming practices. When a program is broken into several modules, each module can be modified and compiled independently. In such a case, these modules have to be linked together to create a complete application. This job is done by a tool known as *linker*. A linker is a program that links several object modules and libraries to form a single, coherent program (executable). Object modules are the machine code output from an assembler or compiler and contain executable machine code and data, together with information that allows the linker to combine the modules together to form a program.

Generally, all HLLs use some in-built functions like calculating square roots, finding logarithm values, and so on. These functions are usually provided by the language itself, the programmer does not need to code them separately. During the program execution process, when a program invokes any in-built function, the linker transfers the control to that program where the function is defined, by making the addresses of these functions known to the calling program.



**Loader** Loaders are a part of the operating system that brings an executable file residing on disk into memory and starts it running. It is responsible for loading, linking, and relocation. In computing, a loader program is a program that performs the functions of a linker program and then immediately schedules the executable code for execution, without necessarily creating an executable file as an output. A loader performs four basic tasks as follows:

1. **Allocation:** It allocates memory space for the programs.
2. **Linking:** It combines two or more separate object programs and supplies the information needed to allow references between them.
3. **Relocation:** It prepares a program to execute properly from its storage area.
4. **Loading:** It places data and machine instructions into the memory.

There are two types of loaders:

1. **Absolute Loader:** It loads the file into memory at the location specified by the beginning portion (header) of the file and then passes control to the program. If the memory space specified by the header is currently in use, execution cannot proceed, and the user must wait until the requested memory becomes free. Moreover, this type of loader performs only loading function. It does not perform linking and program relocation.
2. **Relocating Loader:** This loader loads the program in memory, altering the various addresses as required to ensure correct referencing. The decision as to where in memory the program is placed is done by the operating system, not the file's header. It is a more efficient loader, but there is a slight overhead in terms of a small delay while all the relative offsets are calculated. The relocating loader can only relocate code that has been produced by a linker capable of producing relative code.

## 4.8 FEATURES OF A GOOD PROGRAMMING LANGUAGE

The features of one programming language may differ from the other. One can be easy and simple, while another can be difficult and complex. We judge the success and strength of a programming language with respect to standard features. To begin the language selection process, it is important to establish some criteria for what makes a language good.

**Ease of Use** The language should be easy in writing codes for the programs and executing them. The ease and clarity of a language depends upon its syntax. It should be capable enough to provide clear, simple, and unified set of concepts. The vocabulary of the language should resemble English (or some other natural language). Symbols, abbreviations, and jargon should be avoided unless they are already known to most people. Any concept that cannot easily be explained to amateurs should not be included.

Simplicity helps in the readability of the language. A simple language is easier to grasp and code. Developing and implementing a compiler or interpreter is also easier for simple languages as compared to complex ones.

**Portability** The language should support the construction of code in a way that it could be distributed across multiple platforms (operating systems). Computer languages should be independent of any particular hardware or operating system, that is, programs written on one platform should be able to be transferred to any other computer or platform and there it should perform accurately.

**Naturalness for the Application** The language should have a syntax that allows the program structure to show the underlying logical structure of an algorithm. A programming language should provide a conceptual framework for thinking through algorithms and means of expressing those algorithms through flowcharts.

The advantage of displaying a program in algorithms and flowcharts is that even a novice can understand them easily without learning any programming language.

**Reliability** The language should support the construction of components that perform their intended functions in a satisfactory manner throughout its lifetime. Reliability is concerned with making a system failure free, and thus is concerned with all possible errors. The language should have the support of error detection as well as prevention. It should detect some kinds of specific errors. For example, some errors can be prevented by a strict syntax checking. Also, the language should also be able to detect and report errors in the program. For example, arithmetic overflow and assertions should be detected properly and reported to the programmers immediately so that the error can be rectified. The language should provide reliability by supporting explicit mechanisms for dealing with problems that are detected when the system is in operation (exception handling).

**Safety** It is concerned with the extent to which the language supports the construction of safety-critical systems, yielding systems that are fault-tolerant, fail-safe, or robust in the face of systemic failures. The system must always do what is expected and be able to recover from any situation that might lead to a mishap or actual system hazard. Thus, safety tries to ensure that any failure that results in minor consequences, and even potentially dangerous failures are handled in a fail-safe fashion. Language can facilitate this through such features as built-in consistency checking and exceptional handling.

**Performance** By performance, we mean that the language should not only be capable of interacting with the end users, but also with the hardware. The language should also support software engineering mechanism, discouraging or prohibiting poor practices and supporting maintenance activities. Nowadays, the hardware has become very sophisticated and quiet fast. Hence, the application developed using a good language should tap the maximum resources of the available hardware power in terms of speed and memory efficiency.

**Cost** Cost component is a primary concern before deploying a language at a commercial level. It includes several costs, such as:

- Program execution and translation cost.
- Program creation, testing, and use cost.
- Program maintenance cost.

**Promote Structural Programming** We know that structural programming is the best way to create an application. Hence, a good language should be capable of supporting structural programming. Since, by nature, structural programming facilitates ease in understanding the code, a program is easier to create, debug, and maintain. A structured program also helps programmers to visualize the problem in a logical way, thereby reducing the probability of errors in the code.

**Compact Code** A language should promote compact coding, that is, the intended operations should be coded in a minimum number of lines. Even if the language is powerful but is not able to perform the tasks in small amount of code, then it is of no use. The main reason behind this is that larger code requires more testing and developing time, thereby increasing the cost of developing an application.

**Maintainability** Creating an application is not the end of the system development. It should be maintained regularly so that it can be easily modified to satisfy new requirements or to correct deficiencies. Maintainability is actually facilitated by most of the languages, which makes it easier to understand and then change

the software. Maintainability is closely linked with the structure of the code. If the original code is written in an organized way (structural programming) then it would be easy to modify or add new changes.

**Reusability** The language should facilitate the adaptation of code for the use in other applications. Code is reusable when it is independent of other codes. It is very common, for example, to reuse common data structures, such as stacks, queues, and trees. When these have been defined with common operations on the structures, these abstract data types are easy to reuse.

**Provides Interface to Other Language** From the perspective of the language, interface to other language refers to the extent to which the selected language supports interfacing feature to other languages. This type of support can have a significant impact on the reliability of the data, which is exchanged between two applications, developed with different languages. In case of data exchange between units of different languages, without specific language support, no checking may be done on the data or even on their existence. Hence, the potential for unreliability becomes high.

**Concurrency Support** Concurrency support refers to the extent to which inherent language supports the construction of code with multiple threads of control (also known as *parallel processing*). For some applications, multiple threads of control are very useful or even necessary. This is particularly true for real-time systems and those running on architecture with multiple processors. Concurrency is rarely supported by a language directly; however, language support can make concurrent processing more straightforward and understandable, and it can also provide the programmer with more control over its implementation.

**Standardization** Standardization means the extent to which the language definition has been formally standardized (by recognized bodies, such as ANSI and ISO) and the extent to which it can be reasonably expected that this standard will be followed in a language translator. Non-standard languages may soon become obsolete, rendering it inferior and difficult to use, producing inferior code, which is difficult to maintain. Lack of appropriate support compromises developer productivity and system quality. It also compromises the ease of development, as well as performance and reliability of the code. If the reliability of the code is compromised, not only the system will perform below expectations, but it will also become much more costly during its lifetime.

## SUMMARY

1. A computer needs to be instructed to perform all its tasks. These instructions are provided in the form of *computer program*.
2. *Algorithm* is a precise set of instructions, which specify how to solve a problem. An algorithm must be unambiguous and it should reach a result after a finite number of steps. Algorithms can be converted into programs, flowcharts, and pseudocodes. *Flowchart* is a means of visually presenting the flow of control through the system, the operations performed within the system, and the sequence in which they are performed. By visualizing the process, a flowchart can quickly help identify bottlenecks or inefficiencies where the process can be streamlined or improved. *Pseudocode* is a generic way of describing an algorithm without the use of any specific