ELEMENTARY PROGRAMMING PRINCIPLES

Definition of terms

Computer Program:
A computer program is a set of coded instructions given to the computer, and represents a logical solution to a problem. It directs a computer in performing various operations/tasks on the data supplied to it.

Computer programs may be written by the hardware manufacturers, Software houses, or a programmer to solve user problems on the computer.

Programming:

Programming is the process of designing a set of instructions (computer programs) which can be used to perform a particular task or solve a specific problem.

It involves use of special characters, signs and symbols found in a particular programming language to create computer instructions.

The programming process is quite extensive. It includes analyzing of an application, designing of a solution, coding for the processor, testing to produce an operating program, and development of other procedures to make the system function.

The program created must specify in detail the logical steps to be taken & the method of processing the data input into the computer in order to carry out the specified task.

A computer program performs the following:
1. Accepts data from outside the computer as its input.
2. Carries out a set of processes on the data within the computer memory.
3. Presents the results of this processing as its output, and
4. Stores the data for future use.

Programming Languages:

A programming language is a set of symbols (a language) which a computer programmer uses to solve a given problem using a computer.

The computer must be able to translate these instructions into machine-readable form when arranged in a particular sequence or order.

TERMS USED IN COMPUTER PROGRAMMING

Source program (source code)
The term Source program refers to program statements that the programmer enters in the program editor window, and which have not yet been translated into machine-readable form.

Source code is the code understood by the programmer, and is usually written in high-level language or Assembly language.

Object code (object program).
The term Object code refers to the program code that is in machine-readable (binary) form. This is the code/language the computer can understand, and is produced by a Compiler or Assembler after translating the Source program into a form that can be readily loaded into the computer.
LANGUAGE TRANSLATORS

A computer uses & stores information in binary form, and therefore, it cannot understand programs written in either high-level or low-level languages. This means that, any program code written in Assembly language or high-level language must be translated into Machine language, before the computer can recognize & run these programs.

A Translator is special system software used to convert the Source codes (program statements written in any of the computer programming languages) to their Object codes (computer language equivalents).

The Translators reside in the main memory of the computer, and use the program code of the high-level or Assembly language as input data, changes the codes, and gives the output program in machine-readable code.

In addition, translators check for & identify some types of errors (e.g., Syntax/grammatical errors) that may be present in the program being translated. They will produce error messages if there is a mistake in the code.

Each language needs its own translator. Generally, there are 3 types of language translators:

1. Assembler.
2. Interpreter.
3. Compiler.

Note. Interpreters & Compilers translate source programs written in high-level languages to their machine language equivalents.

Assembler

An assembler translates programs written in Assembly language into machine language that the computer can understand and execute.

Functions of an Assembler.

1). It checks whether the instructions written are valid, and identifies any errors in the program.

The Assembler will display these errors as well as the complete source and object programs. If the program has no errors, the job control will let it run immediately, or save the object program so that it may run it later without translating it again.

2). It assigns memory locations to the names the programmer uses.

E.g., the Assembler keeps a table of these names so that if an instruction refers to it, the Assembler can easily tell the location to which it was assigned.

3). It generates the machine code equivalent of the Assembly instructions.

Usually, the Assembler generates a machine code only when no errors are detected. Some of the errors include;

- Typing mistakes.
- Using the wrong format for an instruction.
- Specifying a memory location outside the range 0 – 2047.

Note. The Assembler cannot detect Logic errors. The programmer knows of these errors only when the program is run & the results produced are incorrect (not what the programmer expected). The programmer must therefore, go through the program & try to discover why an incorrect result was being produced.

Interpreter

An interpreter translates a source program word by word or line by line. This allows the CPU to execute one line at a time.

The Interpreter takes one line of the source program, translates it into a machine instruction, and then it is immediately executed by the CPU. It then takes the next instruction, translates it into a machine instruction, and then the CPU executes it, and so on.
The translated line is not stored in the computer memory. Therefore, every time the program is needed for execution, it has to be translated.

**Compiler**

A compiler translates the entire/whole source program into object code at once, and then executes it in machine language code. These machine code instructions can then be run on the computer to perform the particular task as specified in the high-level program.

The process of translating a program written in a high-level source language into machine language using a compiler is called **Compilation**.

For a given machine, each language requires its own Compiler. E.g., for a computer to be able translate a program written in FORTRAN into machine language; the program must pass through the FORTRAN compiler (which must ‘know’ FORTRAN as well as the Machine language of the computer).

The object code file can be made into a fully executable program by carrying out a **Linking** process, which joins the object code to all the other files that are needed for the execution of the program. After the linking process, an executable file with an .EXE extension is generated. This file is stored on a storage media.

**Points to note.**

- The job of a Compiler is much more difficult than that of an Assembler in that, a single statement in a high-level language is equivalent to many machine instructions.
- The format of an Assembly instruction is fairly fixed, while high-level languages give a lot of freedom in the way the programmer writes statements.

**Functions of a compiler.**

A Compiler performs the following tasks during the compilation process:

1. It identifies the proper order of processing, so as to execute the process as fast as possible & minimize the storage space required in memory.
2. It allocates space in memory for the storage locations defined in the program to be executed.
3. It reads each line of the source program & converts it into machine language.
4. It checks for **Syntax errors** in a program (i.e., statements which do not conform to the grammatical rules of the language). If there are no syntax errors, it generates machine code equivalent to the given program.
5. It combines the program (machine) code generated with the appropriate subroutines from the library.
6. It produces a listing of the program, indicating errors, if any.

**Differences between Compilers and Interpreters**

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Compiler</th>
</tr>
</thead>
</table>
| 1. Translates & executes each statement of the source code one at a time.  
The source code instruction is translated & immediately obeyed by the computer hardware before the next instruction can be translated. (Translation & execution go together). | 1. Translates all the source code statements at once as a unit into their corresponding object codes, before the computer can execute them.  
A Compiler translates the entire source program first to machine code, and then the code is executed by the CPU. (Translation & execution are separate phases) |
2. Translates the program each time it is needed for execution; hence, it is slower than compiling.

3. Interpreted object codes take less memory compared to compiled programs.

4. For an Interpreter, the syntax (grammatical) errors are reported & corrected before the execution can continue.

5. An Interpreter can relate error messages to the source program, which is always available to the Interpreter. This makes debugging of a program easier when using an Interpreter than a Compiler.

2. Compiled programs (object codes) can be saved on a storage media and run when required; hence executes faster than interpreted programs.

3. Compiled programs require more memory as their object files are larger.

4. For a Compiler, the syntax errors are reported & corrected after the source code has been translated to its object code equivalent.

5. Once the source program has been translated, it is no longer available to the Compiler, so the error messages are usually less meaningful.

Linkers & Loaders

Computer programs are usually developed in Modules or Subroutines (i.e., program segments meant to carry out the specific relevant tasks). During program translation, these modules are translated separately into their object (machine) code equivalents.

The Linker is a utility software that accepts the separately translated program modules as its input, and logically combines them into one logical module, known as the Load Module that has got all the required bits and pieces for the translated program to be obeyed by the computer hardware.

The Loader is a utility program that transfers the load module (i.e. the linker output) into the computer memory, ready for it to be executed by the computer hardware.

Syntax

Each programming language has a special sequence or order of writing characters. The term Syntax refers to the grammatical rules, which govern how words, symbols, expressions and statements may be formed & combined.

Semantics

These are rules, which govern the meaning of syntax. They dictate what happens (takes place) when a program is run or executed.

Review Questions.

1. Define the following terms:
   (i). Computer program.
   (ii). Programming.
   (iii). Programming language.

2. With reference to programming, distinguish between Source program and Object code.

3. What is the function(s) of: Assemblers, Interpreters and Compilers in a computer system?

4. (a). What are the main functions of a Compiler?
   (b). Differentiate between a Compiler and an Interpreter.
LEVELS OF PROGRAMMING LANGUAGES

There are many programming languages. The languages are classified into 2 major categories:
1). Low-level programming languages.
2). High-level programming languages.

Each programming language has its own grammatical (syntax) rules, which must be obeyed in order to write valid programs, just as a natural language has its own rules for forming sentences.

LOW-LEVEL LANGUAGES

These are the basic programming languages, which can easily be understood by the computer directly, or which require little effort to be translated into computer understandable form.

They include:
2. Assembly languages.

Features of low-level languages
– They are machine hardware-oriented.
– They are not portable, i.e., a program written for one computer cannot be installed and used on another computer of a different family.
– They use Mnemonic codes.
– They frequently use symbolic addresses.

Machine languages (1st Generation languages)

Machine language is written using machine codes (binary digits) that consist of 0’s & 1’s.

The computer can readily understand Machine code (language) instructions without any translation.

A programmer is required to write his program in strings of 0’s & 1’s, calculate & allocate the core memory locations for his data and/or instructions.

Different CPU’s have different machine codes, e.g., codes written for the Intel Pentium processors may differ from those written for Motorola or Cyrix processors. Therefore, before interpreting the meaning of a particular code, a programmer must know for which CPU the program was written.

A machine code instruction is made up of 2 main parts;
(i). An Address (operand):
   It specifies the location (address) of the computer memory where the data to be worked upon can be found.
(ii). A Function (operation) code:
   It states to the Control Unit of the CPU what operation should be performed on the data/item held in the address, e.g., Addition, Subtraction, Division, Multiplication, etc.

Note. The computer can only execute instructions which are written in machine language. This is because; it is the only language which the computer can understand. Therefore, any program written in any other programming language must first be translated into machine language (binary digits) before the computer can understand.
Assembly language (2nd Generation Languages).

Assembly languages were developed in order to speed up programming (i.e., to overcome the difficulties of understanding and using machine languages).

The vocabulary of Assembly languages is close to that of machine language, and their instructions are symbolic representations of the machine language instructions.

- Assembly language programs are easier to understand, use & modify compared to Machine language programs.
- Assembly language programs have less error chances.

To write program statements in Assembly language, the programmer uses a set of symbolic operation codes called Mnemonic codes.

The code could be a 2 or 3 shortened letter word that will cause the computer to perform specific operation. E.g., MOV – move, ADD - addition, SUB – subtraction, RD - read.

Example;

RD PAT, 15 (read the value 15 stored in the processor register named PAT)
SUB PAT, 10 (subtract 10 from the value in register PAT)

A program written in an Assembly language cannot be executed/obeyed by the computer hardware directly. To enable the CPU understand Assembly language instructions, an Assembler (which is stored in a ROM) is used to convert them into Machine language.

The Assembler accepts the source codes written in an Assembly language as its input, and translates them into their corresponding computer language (machine code/ object code) equivalent.

Comments are incorporated into the program statements to make them easier to be understood by the human programmers.

Assembly languages are machine-dependent. Therefore, a program written in the Assembly language for a particular computer cannot run on another make of computer.

Advantages of Low-level languages

1. The CPU can easily understand machine language without translation.
2. The program instructions can be executed by the hardware (processor) much faster. This is because; complex instructions are already broken down into smaller simpler ones.
3. Low-level languages have a closer control over the hardware, are highly efficient & allow direct control of each operation.
   They are therefore suitable for writing Operating system software & Game programs, which require fast & efficient use of the CPU time.
4. They require less memory space.
5. Low-level languages are stable, i.e., they do not crash once written.

Disadvantages of Low-level languages

Very few computer programs are actually written in machine or Assembly language because of the following reasons;

1. Low-level languages are difficult to learn, understand, and write programs in them.
2. Low-level language programs are difficult to debug (remove errors from).
3. Low-level languages have a collection of very detailed & complex instructions that control the internal circuiting of the computer. Therefore, it requires one to understand how the computer codes internally.
4. Relating the program & the problem structures is difficult, and therefore cumbersome to work with.
5. The programs are very long; hence, writing a program in a low-level language is usually tedious & time consuming.
6. The programs are difficult to develop, maintain, and are also prone to errors (i.e., it requires highly trained experts to develop and maintain the programs).

7. Low level languages are machine-dependent (specific), hence non-portable.
   This implies that, they are designed for a specific machine & specific processor, and therefore, cannot be transferred between machines with different hardware or software specifications.

8. It is not easy to revise the program, because this will mean re-writing the program again.

HIGH-LEVEL PROGRAMMING LANGUAGES

High-level languages were developed to solve (overcome) the problems encountered in low-level programming languages.

The grammar of High-level languages is very close to the vocabulary of the natural languages used by human beings. Hence; they can be read and understood easily even by people who are not experts in programming.

Most high-level languages are general-purpose & problem-oriented. They allow the programmer to concentrate on the functional details of a program rather than the details of the hardware on which the program will run.

High-level language programs are machine-independent, (i.e., they do not depend on a particular machine, and are able to run in any family of computers provided the relevant translator software is installed).

Programs written in a high-level language cannot be obeyed by the computer hardware directly. Therefore, the source codes must be translated into their corresponding machine language equivalent. The translation process is carried out by a high-level language software translator such as a Compiler or an Interpreter.

Features of high-level programming languages.

- They contain statements that have an extensive vocabulary of words, symbols, sentences & mathematical expressions, which are very similar to the normal English language.

  Example;
  
  Read (TaxablePay);
  IF TaxablePay<1000 THEN
  Tax: =0;
  ELSE
  Tax: =TaxRate * TaxablePay;
  Write (Tax: 6:2);

- Allow modularization (sub-routines).
- They are ‘user-friendly’ and problem-oriented rather than machine-based. This implies that, during a programming session, the programmer concentrates on problem-solving rather than how a machine operates.
- They require one to be obey a set of rules when writing the program.
- Programs written in high-level languages are shorter than their low-level language equivalents, since one statement translates into several machine code instructions.
- The programs are portable between different computers.

Purpose of High-level languages.

1. To improve the productivity of a programmer. This is because; the source programs of high-level languages are shorter than the source programs of low-level languages, since one statement translates into several machine code instructions.

2. To ease the training of new programmers, since there is no need to learn the detailed layout of a procession/sequence.
3. To speed up testing & error correction.
4. To make programs easy to understand & follow.

**Advantages of High-level languages.**
1. They are easily portable, i.e., they can be transferred between computers of different families and run with little or no modification.
2. High-level language programs are short, and take shorter time to be translated.
3. They are easy to lean, understand and use.
4. They are easy to debug (correct/remove errors), & maintain.
5. High level language programs are easy to modify, and also to incorporate additional features thus enhancing its functional capabilities.
6. They are ‘user-friendly’ & problem-oriented; hence, can be used to solve problems arising from the real world.
7. They enable programmers to adapt easily to new hardware. This is because; they don’t have to worry about the hardware design of the computer.
8. High-level language programs are self-documenting, i.e., the program statements displays the transparency of purpose making the verification of the program easy.
9. High level languages are more flexible; hence, they enhance the creativity of the programmer and increase his/her productivity in the workplace.

**Disadvantages of using High-level languages**
1. High-level languages are not machine-oriented; hence, they do not use of the CPU and hardware facilities efficiently.
2. The languages are machine-independent, and cannot be used in programming the hardware directly.
3. Each high-level language statement converts into several machine code instructions. This means that, they use more storage space, and it also takes more time to run the program.
4. Their program statements are too general; hence, they execute slowly than their machine code program equivalents.
5. They have to be interpreted or compiled to machine-readable form before the computer can execute them.
6. The languages cannot be used on very small computers.
   The source program written in a high-level language needs a Compiler, which is loaded into the main memory of the computer, and thus occupies much of memory space. This greatly reduces the memory available for a source program.

**TYPES OF HIGH-LEVEL LANGUAGES.**
High-level languages are classified into five different groups:
1. Third generation languages (Structured / Procedural languages).
2. Fourth generation languages (4GLs).
3. Fifth generation languages (5GLs)
4. Object-oriented programming languages (OOPs).
5. Web scripting languages.

The various types of high-level languages differ in:
- The data structures they handle.
- The control structures they support.
- The assignment instructions they use.
- Application areas, e.g., educational, business, scientific, etc.
STRUCTURED LANGUAGES

A structured (procedural) language allows a large program to be broken into smaller sub-programs called *modules*, each performing a particular (single) task. This technique of program design is referred to as *structured programming*.

Structured programming also makes use of a few simple control structures in problem solving. The 3 basic control structures are:

- Sequence
- Selection.
- Iteration (looping).

**Advantages of structured programming.**

1. It is flexible.
2. Structured programs are easier to read.
3. Programs are easy to modify because; a programmer can change the details of a section without affecting the rest of the program.
4. It is easier to document specific tasks.
5. Use of modules that contain standard procedures throughout the program saves development time.
6. Modules can be named in such a way that, they are consistent and easy to find in documentation.
7. Debugging is easier because; each module can be designed, coded & tested independently.

Examples of Third generation programming languages include:

1). **BASIC** *(Beginners All-purpose Symbolic Instructional Code)*.
   
   BASIC is a simple general-purpose high-level language used in most computer processing tasks such as developing business and educational applications.
   
   It is easy to learn & use; hence, suitable for students who wish to easily learn programming. Translation in most versions of BASIC is carried out by an *Interpreter*.
   
   **Disadvantages of BASIC.**
   
   - BASIC is available in so many versions with different dialects/languages & therefore, it has no standard.
   - Some dialects are limited to data & control structures they support.
   - Some versions of BASIC offer limited facilities in terms of structured programming & meaningful variable names.

2). **PASCAL**
   
   PASCAL is a general-purpose, high-level programming language, which was named after a French mathematician called *Blaise Pascal*.
   
   It was developed as an academic tool to help in the teaching and learning of structured programming.
   
   PASCAL supports structured programming, i.e., it uses procedures & functions, which allow a *top-down* approach to solving problems.
   
   - It is not easy to learn because; it has strict rules in its grammar on typing of variables (data names) & declarations.
   - It is poor (has limited ability) on handling of data files.
3). **COBOL (COmmon Business Oriented Language)**

COBOL is designed for developing programs that solve business problems, e.g., can be used to develop commercial data processing applications such as computer-based inventory control systems.

COBOL is mostly used where large amounts of data are to be handled, because it supports powerful data & control structures.

COBOL programs are *semi-compiled*, and the intermediate code is *interpreted*.

A program written in COBOL language consists of 4 divisions:

(i). **Identification division**: Where the programmer & the program details are specified, e.g., program ID, programmer name, etc.

(ii). **Environment division**: Where the equipments to be used by the source & the object programs are defined, e.g., the computer hardware.

(iii). **Data division**: Where the various files to be used by the program are described, e.g., a description of the input files.

(iv). **Procedure division**: Where all the procedures required to manipulate/interrelate the data into information are defined.

**Advantages of COBOL.**

- It is easy to read.
- It is portable, i.e., can be used on different types of computers. This is because; it has an *American National Institute* standard.

  **American National Standards Institute (ANSI):** – An international organization that devised/invented the group of standardized symbols used in flowcharting.

- It is widely used, and has a pool of skilled programmers.

**Disadvantage of COBOL.**

- The structure of a COBOL program is too long even for simple programs.

  E.g., consider the following assignment statement:

  \[
  \text{DIVIDE } A \text{ into } B \text{ giving } C. 
  \]

  This statement when used in BASIC language can much short ‘\( C = A / B \)’. However, notice that the COBOL statement above is more self defining.

4). **FORTRAN (FORmula TRANslator)**

It was developed for mathematicians, scientists and engineers. It provides an easier way of writing scientific & engineering applications.

FORTRAN statements are mostly in form of mathematical expressions; hence, it is useful in writing of programs that can process numeric data.

FORTRAN programs are *compiled*.

**Advantages of FORTRAN.**

- It is portable, i.e. it can be used on different types of computers.

**Disadvantage of FORTRAN.**

- It is not suited for business applications.

5). **Ada**

This language was named after the first lady programmer Ada Lovelace.

It is suitable for developing *military, industrial* and *real-time* systems.
6). **C**

C is mainly used for developing system software such as the operating system as well as developing the application packages. It has powerful commands that permit the rapid development of programs, and allows direct control over the hardware.

**Disadvantage of C**

– It is difficult to read & learn because of its strict dialect rules.

7). **LOGO**

LOGO was designed for educational use in which children can explore & develop concepts through programming the movement of a pen.

8). **COROL**

COROL is used in Real-time processing.

COROL programs are compiled.

9). **RPG** *(Report Program Generator)*

RPG is used in report generating applications, (i.e. it is designed to facilitate the output of reports of business data).

A **Report generator** is a software tool that extracts stored data to create customized reports that are not normally/usually produced by existing applications.

10). **SNOBOL** *(String Oriented Symbolic Language).*

It is a high-level language designed to manipulate strings of characters. It is therefore used for non-numeric applications.

**FOURTH GENERATION LANGUAGES (4GL’S).**

4GLs make programming even easier than the 3GLs because; they present the programmer with more programming tools, such as command buttons, forms, textboxes etc. The programmer simply selects graphical objects called **controls** on the screen, and then uses them to create designs on a form by dragging a mouse pointer.

The languages also use **application generators** (which in the background) to generate the necessary program codes; hence, the programmer is freed from the tedious work of writing the code.

4GLs are used to enquire & access the data stored in database systems; hence, they are described as the **Query languages**.

**Purpose of fourth generation languages.**

The 4GL’s were designed to meet the following objectives: -

1. To speed up the application-building process, thereby increasing the productivity of a programmer.
2. To enable quick & easy amendments and alteration of programs.
3. To reduce development & maintenance costs.
4. To make languages user-friendly. This is because, the 4GL’s are designed to be user-oriented, unlike the 3rd generation languages which are problem & programmer oriented.
5. To allow non-professional end-users to develop their own solutions.
6. To generate bug-free codes from high-level expressions of requirements.
Examples of 4GLs are:
- Visual Basic
- Delphi Pascal
- Visual COBOL (Object COBOL)
- Access Basic

Advantages of fourth generation languages.
1. They are user-based, and therefore, easy to learn & understand.
2. The grammar of 4GL’s is very close to the natural English language. It uses menus & prompts to guide a non-specialist to retrieve data with ease.
3. Very little training is required in order to develop & use 4GL programs.
4. They provide features for formatting of input, processing, & instant reporting.

FIFTH GENERATION LANGUAGES (5GL’S).
The 5GL’s are designed to make a computer solve a problem by portraying human-like intelligence.
The languages are able to make a computer solve a problem for the programmer; hence, he/she does not spend a lot of time in coming up with the solution. The programmer only thinks about what problem needs to be solved and what conditions need to be met without worrying about how to implement an algorithm to solve the problem.
5GLs are mostly used in artificial intelligence.
Examples of 5GLs are:
- PROLOG
- LISP
- Mercury
- OCCAM.

1). **LISP (LIST Processing)**
   In LISP, both programs & data are arranged (structured) as lists.
   It is used in artificial intelligence. However, it is not suitable for commercial data processing applications.

2). **PROLOG (PROGRAM in LOGic)**
   PROLOG was developed from LISP by the Japanese.
   It is designed for use with Expert systems & Artificial Intelligence. It is mostly used for solving problems, which involve objects and relationships between objects.
   Like LISP, it is not suitable for commercial data processing applications.

OBJECT-ORIENTED PROGRAMMING LANGUAGES (OOPs)
Object-Oriented Programming is a new approach to software development in which data & procedures that operate on data are combined into one object.
OOPs use objects. An Object is a representation of a software entity such as a user-defined window or variable. Each object has specific data values that are unique to it (called state) and a set of the things it can accomplish called (functions or behaviour).
Several objects can be linked together to form a complete program. Programs send messages to an object to perform a procedure that is already embedded in it. This process of having data and functions that operate on the data within an object is called encapsulation.
The data structure & behaviour of an object is specified/described by a template (called a class). Classes are hierarchical, and it is possible to pass the data & behaviour of an object in one class
down the hierarchy.

Object-Oriented programming enables rapid program development. Every object has properties such as **colour**, **size**, **data source**, etc, which can be set easily without much effort. In addition, every object has events associated with it that can be used to trigger certain actions, e.g. remove the window from the screen on clicking the ‘Close’ button.

OOP has contributed greatly to the development of *graphical user interface operating systems* and *application programs*.

Examples of Object-oriented programming languages are:

- Simula
- C++
- SmallTalk
- Java
  
  Java is sometimes associated with development of websites, but it can be used to create whole application programs that do not need a web browser to run.

**JAVA**

Java is an OOP language that resembles **Object C** (a simplified form of C++).

The code of Java displays graphics, accesses the network, and interfaces with users via a set of capabilities known as **classes**. Classes define similar states & common methods for the behavior of an object.

JAVA programs are not compiled into machine code; instead, they are converted into a collection of bytes that represent the code for an abstract *Java Virtual machine (VM)*. A Java interpreter running on a physical machine is then used to translate those bytes into local actions, such as printing a string or drawing a button.

**WEB SCRIPTING LANGUAGES.**

Web scripting languages are mostly used to create or add functionalities on web pages.

*Web pages* are used for creating Web sites on the Internet where all sorts of advertising can be done.

Web pages are hypertext (plain-text) documents written using a language called *HyperText Markup Language (HTML)*. HTML documents have a file extension of `.Html` or `.Htm`.

**Note.** HTML doesn’t have the declaration part and control structures, and has many limitations. Therefore, to develop functional websites, it must be used together with other web scripting languages like *JavaScript*, *VBScript* and *Hypertext Preprocessor*.

**Comparison of Programming languages.**

<table>
<thead>
<tr>
<th><strong>Machine language</strong></th>
<th><strong>Assembly language</strong></th>
<th><strong>High-level languages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Instruction set is made up of binary digits (0’s &amp; 1’s).</td>
<td>1. Instruction set is made up of Mnemonics &amp; labels.</td>
<td>1. Instruction set is similar to English language statements &amp; mathematical operators.</td>
</tr>
<tr>
<td>2. Instruction is made of 2 parts: <em>operation code &amp; operand</em>.</td>
<td>2. Instruction is made up of 2 parts: <em>operation code &amp; operand</em>, but comments can be added.</td>
<td>2. The instruction varies depending on the particular language.</td>
</tr>
<tr>
<td>3. No translation is needed. (This is the computer language; hence, the computer understands it directly).</td>
<td>3. Uses an Assembler to convert the assembly language source codes to their object code equivalents</td>
<td>3. Uses compiler or interpreter <em>Compiler</em> translates all the source code at once into object code; <em>Interpreter</em> translates line by line.</td>
</tr>
</tbody>
</table>
### Factors to consider when choosing a Programming language.

The following factors should be considered when choosing a Programming language to use in solving a problem:

1. The availability of the relevant translator
2. Whether the programmer is familiar with the language
3. Ease of learning and use
4. Purpose of the program, i.e., application areas such as education, business, scientific, etc.
5. **Execution time**
   - Applications that require quick response are best programmed in machine code or assembly language. High-level languages are not suitable for such application because, they take long to be translated & executed.

6. **Development time**
   - **Development time** is the time a programmer takes to write and run a program.
   - High-level languages are easy to read, understand and develop; hence, they require less development time. Machine code & Assembly languages are relatively difficult to read, understand and develop; hence, they are time-consuming.

7. **Popularity**
   - The language selected should be suitable and/or successful in the market with respect to the problems to be solved.

8. **Documentation**
   - It should have accompanying documentation (descriptions) on how to use the language or maintain the programs written in the language.
9). Maintenance

Programs are developed to solve specific problems, and the problems keep on changing; hence, the programs are also changed to perform the new functions.

**Program maintenance** is the activity of incorporating more routines onto the program, modifying the existing routines or removing the obsolete routines to make the program adapt to a functionally enhanced environment.

The maintenance is made easier if the language used is easy to read and understand.

10). Availability of skilled programmers

The language selected should have a pool of readily available programmers to ease the programming activity, and reduce development time.

Review Questions

1. (a). What is a Programming language?
   (b). Explain the two levels of programming languages.

2. (a). What is meant by ‘Machine language’?
   (b). Explain why machine language programming is so error-prone.
   (c). Show the difference between Machine language and Assembly language.
   (d). Give two advantages & three disadvantages of Machine language programming.

3. (a). What are High-level languages?
   (b). Give the features/characteristics of high-level programming languages.
   (c). Describe briefly how a program written in high-level programming language becomes a machine code program ready for operational use.
   (d). Explain the advantages and disadvantages of using a High-level programming language for writing a program.
   (e). List four examples of high-level programming languages. Indicate the application of each language in computing.

4. (a). What is meant by *program portability*?
   (b). Why are low-level languages not considered to be portable?

5. List 8 factors that need to be considered when selecting a programming language.
PROGRAM DEVELOPMENT.

Stages involved in the program development cycle.

The process of program development can be broken down into the following stages:
1. Problem recognition (Identification of the problem).
2. Problem definition.
3. Program design.
4. Program coding.
5. Program testing & debugging.
6. Program Implementation and maintenance.
7. Program documentation.

Problem recognition.

*Problem recognition* refers to the understanding and interpretation of a particular problem.

The programmer must know what problem he/she is trying to solve. He/she must also understand clearly the nature of the problem & the function of the program.

In order to understand a problem, look for the keywords such as *compute, evaluate, compare,* etc.

Usually, a programmer identifies problems in the environment and tries to solve them by writing a computer program.

There are 3 situations that cause the programmer to identify a problem that is worth solving:
1. *Problems* or undesirable situations that prevent an individual or organizations from achieving their purpose.
2. *Opportunity* to improve the current program.
3. A *new directive* given by the management requiring a change in the current system.

*Sample problem:* Develop a program that can be used to calculate/find the area of a circle. Use the equation \( A = \pi \times r^2 \).

Problem definition (Problem Analysis).

In Problem definition, the programmer tries to define (determine) the:
- (i). Output expected from the program.
- (ii). Inputs needed to generate the output information.
- (iii). Processing activities (requirements), and
- (iv). Kind of files which may be needed.

The programmer should write a narrative on what the program will do, and how it is meant to achieve the intended purpose. Within this narrative, he/she is required to determine what data is to be input & what information is to be output.

*For example:*

In calculating the area of any circle, the parameters needed to determine the area of any circle are:

1. **Input:** (a) Pie \( (\pi) \) which is a constant.
   (b) The radius of the circle.

2. **Process:** The formula for calculating area of a circle, which is \( \pi \times \text{radius} \times \text{radius} \).

3. **Output:** The area of the circle (A).

At the end of the problem definition, the programmer is required to write a requirements report/document for the new program. This document will enable the programmer to come up with a program design that meets the needs at hand.
Note. Problem definition should be done thoroughly to ensure user satisfaction, and to facilitate the subsequent stages in the program development cycle. A failure at this stage usually results in a system that will not work as intended, or that may not work at all.

Program design

Program design is the actual development of the program’s process or problem solving logic called the Algorithm.

It involves identifying the processing tasks required to be carried out in order to solve the problem.

The design stage enables the programmer to come up with a model of the expected program (or a general framework (outline) of how to solve the problem, and where possible, break it into a sequence of small & simple steps.

The models show the flow of events throughout the entire program from the time data is input to the time the program gives out the expected information.

- The processing tasks must be in order & systematic. Therefore, the programmer identifies the processing tasks required, and the exact order in which they are to be carried out.
- The design process does not take account of the programming language to be used in the final product, since it only defines program logic.
- Program design provides for easy maintenance.

Note. It is important to design programs before entering them into the computer. The programmer should only attempt to covert a design into a program code after ensuring that it is logically correct. If possible, check the logical order on the desk.

Some programmers produce rough & ready solutions at a Keyboard, and continue to amend the programs until eventually the program appears to do what was expected. This is not recommended in programming because of the following reasons:

1. The final code may not be easy to follow, since it was just cobbled together.
2. Variable names & specific items of code may not be documented.
3. Programs produced by continuous amendments & changing of codes mostly lead to unforeseen side effects.
   E.g., there may not have been plan for testing the program or procedures, hence, the program may easily fail.
4. A programmer may be asked to modify the code at a later date. Without sufficient documentation, the programmer will be forced to trace through the program in order to gain an insight into how the program functions.

Modular programming

Many programs are non-monolithic (i.e., they are not usually made up of one large block of code). Instead, they are made up of several units called modules, that work together to form the whole program with each module performing a specific task.

This approach makes a program flexible, easier to read, and carry out error correction.

Program coding

Program coding is the actual process of converting a design model into its equivalent program.

Coding requires the programmer to convert the design specification (algorithm) into actual computer instructions using a particular programming language.

For example;

The programmer may be required to write the program code either in Pascal, C++, Visual Basic
or Java, and develop (invent) suitable identifiers, variable names, & their data types. However, remember that, at this stage the coding is still a Pencil & paper exercise.

The end result of this stage is a source program that can be translated into machine readable form for the computer to execute and solve the target problem.

**Rules followed in coding a program.**

1. Use the standard identifiers or reserved words.
2. Make the program more readable by using meaningful identifiers.
3. Don’t use similar variables.
4. Keep spellings as normal as possible.
5. Use comments to explain variables & procedures. This makes the program readable.
6. Avoid tricks – write the program using straightforward codes that people can readily understand.
7. Modularize your program.

**Sample programs written in Pascal language.**

**Example 1:**

Develop a program code that would be used to solve the equation of a straight line given by the expression: \( Y = mx + c \)

Program StraighLine (input, output);

VAR
y, m, x, c: INTEGER;

BEGIN
  Writeln ('Input the value of M');
  Readln (M);
  Writeln ('Input the value of X');
  Readln (X);
  Writeln ('Input the value of C');
  Readln (C);
  Y: = (m * x) + c;
  Writeln ('The value of y is:', Y);
END.

<table>
<thead>
<tr>
<th>Pascal code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program StraighLine (input, output);</td>
<td>This is the program Header.</td>
</tr>
<tr>
<td></td>
<td>The word “Program” indicates the beginning of the program whose name is StraightLine.</td>
</tr>
<tr>
<td></td>
<td>The (input, output) statements shows that the program expects some input from the Keyboard and display the output on the Screen.</td>
</tr>
<tr>
<td>VAR</td>
<td>VAR is short form for Variable. A variable is a location for data in the computer memory.</td>
</tr>
<tr>
<td>y, m, x, c: INTEGER;</td>
<td>This statement tells the computer that variables are about to be declared. When a variable is declared, the computer sets aside some memory space to store a value in the variable.</td>
</tr>
<tr>
<td></td>
<td>Four variables of type Integer have been declared.</td>
</tr>
<tr>
<td></td>
<td>This means that, the memory spaces that will be set</td>
</tr>
</tbody>
</table>
aside can only hold values that are whole numbers.

<table>
<thead>
<tr>
<th><strong>BEGIN</strong></th>
<th>The <code>Begin</code> statement marks the start of the program body. Statements in this section are executed by the computer. E.g., execution starts by asking the user to input the value of <code>m</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>WriteLn ('Input the value of M');</strong></td>
<td>The <code>WriteLn</code> statement displays whatever is between the inverted commas in the brackets. The statements will be sent to the screen exactly the way they appear in the brackets. This is because; the inverted commas are meant to make the output readable on the screen. To display the value held in a variable on the screen, remove the inverted commas and write the name of the variable in the brackets, e.g., <code>WriteLn (y)</code> will display the value held in the variable <code>y</code>.</td>
</tr>
<tr>
<td><strong>Readln (M);</strong></td>
<td>The <code>Read</code> or <code>Readln</code> statement reads a value and stores it in a variable. When the program is running, a <code>Read/Readln</code> statement in the code will display blinking cursor that indicates to the user where to type the input.</td>
</tr>
<tr>
<td><strong>Y: = (m * x) +c;</strong></td>
<td>Calculates the value of <code>y</code>. In Pascal, the symbol <code>:=</code> is called the <strong>Assignment statement</strong>. The values on the right are calculated then the answer stored in the variable <code>y</code> which is on the left of the assignment symbol.</td>
</tr>
<tr>
<td><strong>WriteLn ('The value of y is:', Y);</strong></td>
<td>The <code>WriteLn</code> displays the value held in the variable <code>y</code> on the screen. <strong>Note.</strong> <code>Y</code> is not within the inverted commas.</td>
</tr>
<tr>
<td><strong>END.</strong></td>
<td>The <code>END.</code> statement shows the end of a program.</td>
</tr>
</tbody>
</table>

*Example 2:*

Program AreaCircle (input, output);

```pascal
CONST
  Pi = 3.142;
VAR
Radius, Area: REAL;
BEGIN
  WriteLn ('Enter the radius');
  Readln (Radius);
  Area: = Pi * Radius * Radius;
  WriteLn ('The Area is', Area);
END.
```

<table>
<thead>
<tr>
<th><strong>Pascal code</strong></th>
<th><strong>Explanation</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program AreaCircle (input, output);</td>
<td>The Header of the program.</td>
</tr>
<tr>
<td>The statements in ( ) shows that the user inputs data via Keyboard and the program display information on the Screen.</td>
<td></td>
</tr>
<tr>
<td><strong>CONST</strong></td>
<td>A constant has been declared with a name <code>Pi</code> and value 3.142.</td>
</tr>
<tr>
<td>Pi = 3.142;</td>
<td>Variables with fractional parts have been declared.</td>
</tr>
</tbody>
</table>
Radius, Area: REAL;
BEGIN

Marks the beginning of the program body.

Writeln ('Enter the radius');

Displays on the screen the string between the inverted commas.

Readln (Radius);

Displays a blinking cursor that tells the user that an input is needed before the program can continue.

Area: = Pi * Radius * Radius;

Calculates the Area. An assignment statement (: =) has been used.

Writeln ('The Area is', Area);

Displays the value stored in the variable Area.

END.

Marks the end of the program.

Revision Questions.
1. State the rules followed in coding a program.

Program Testing and Debugging

After designing & coding, the program has to be tested to verify that it is correct, and any errors detected removed (debugged).

TESTING:

Testing is the process of running computer software to detect/find any errors (or bugs) in the program that might have gone unnoticed.

During program testing, the following details should be checked;

- The reports generated by the system.
- The files maintained in connection to the system’s information requirements.
- The input to the system.
- The processing tasks.
- The controls incorporated within the system.

Note. The testing process is a continuous process, and it ends only when the Programmer & the other personnel involved are satisfied that when operational, the program will meet the objectives and the growing demands of the organization.

Types of program errors

There are 5 main types of errors that can be encountered when testing a program. These are:

1. Syntax errors.
2. Run-time (Execution) errors.
3. Logical (arithmetic) errors.
4. Semantic errors.
5. Lexicon errors.

Syntax errors

Every programming language has a well-defined set of rules concerning formal spellings, punctuations, naming of variables, etc. The instructions are accepted only in a specified form & and must be obeyed by the programmer.

Syntax errors are therefore, programming errors/mistakes that occur if the grammatical rules of a particular language are not used correctly.

Examples:

(i). Punctuation mistakes, i.e., if the programmer does not use the right punctuations & spaces needed by the translator program, e.g., omitting a comma or a semicolon.
(ii). Improper naming of variables.
(iii). Wrong spellings of user defined and reserved words.
Reserved words are those words that have a special meaning to the programming language, and should not be used by the programmer for anything else.

Syntax errors are committed by the programmer when developing, or transcribing the program, and can be detected by the language translators, such as the Compiler as it attempts to translate a program. Such errors must be corrected by the programmer before the program runs.

**Logical (arithmetic) errors.**

These are errors in the program logic. Logical errors relate to the logic of processing followed in the program to get the desired results. E.g., they may occur as a result of misuse of logical operators.

Logical errors cannot be detected by the translator. The programmer will detect them when the program results are produced.

The program will run, but give the wrong output or stop during execution.

**Run-time (Execution) errors.**

These errors occur during program execution.

Run-time (execution) errors occur when the programmer introduces new features in the program, which are not part of the translator’s standards.

For example; they may occur if:

(i). The computer is asked to divide a number by zero.

(ii). The number generated as a result of an instruction is too large to fit in a memory location.

(iii). When you raise a number to a very big power that cannot be accommodated in the register’s structure of the computer.

(iv). In case of a closed loop in the program, leading to a set of instructions being executed repetitively for a long time.

Execution errors are not detected by the translator programs, but are detected by the computer during execution. Sometimes, execution errors may lead to premature end of a program.

To detect and eliminate Execution errors, a test run should be performed on the program after it has been translated.

**Semantic errors.**

These are meaning errors. They occur when the programmer develops statements, which are not projecting towards the desired goal. Such statements will create deviations from the desired objectives.

Semantic errors are not detected by the computer. The programmer detects them when the program results are produced.

*Example;*

a). IF GP>=1500 OR 2200 THEN
   TAX: = GP - (GP * 13%)

b). IF GP>=1500 AND GP<= 2200 THEN
   TAX: = GP - (GP * 13%)

In the 1st statement, if the selection is between 1500 & 2200, the computer will pick only 1500 & 2200, and the other values will not be touched.

In the 2nd statement, the computer will be able to pick all the values between 1500 & 2200 because of the ‘AND’ operator.

**Lexicon errors.**

These are the errors, which occur as a result of misusing Reserved words (words reserved for a particular language).
Revision Questions.

1. State the three types of errors that can be experienced in program testing, and how each can be detected.
2. Syntax errors can be detected by the help of translators while logical errors are detected differently. Explain FIVE methods which can be used to detect Logical errors.

DEBUGGING:
The term Bug is used to refer to an error in a computer program.

Most programming errors often remain undetected until an attempt is made to translate a program.

The most common errors include:-
- Improperly declared Constants and Variables.
- A reference to undeclared variable.
- Incorrect punctuation.

Debugging is therefore, the process of detecting, locating & correcting (removing, eliminating) all errors (mistakes or bugs) that may exist in a computer program.

TYPES OF TESTING (Methods of error detection)
For the program to be assumed as correct, several testing needs to be conducted by the programmer to ascertain/establish their validity.

There are several methods of testing a program for errors. These include:
1. Dry running (Desk checking).
2. Translator system checking.
3. Functional testing.
4. Use of Test data.
5. Use of debugging utilities.
6. Diagnostic procedures.
7. System test with actual data.

Dry Running (Desk checking):

Dry running is a method of checking a program for errors by making the corrections on a paper before entering it in the program editor.

It involves going through the program while still on paper verifying & validating its possible results. If the final results agree with the original test data used, the programmer can then type the program into the computer and translate it.

√ Dry running helps the programmer to identify the program instructions, detect the most obvious syntax and logical errors, & the possible output.
√ Dry running is much faster. This is because; it involves the use of human brain as the processor, which has got a well inbuilt common sense.

Translator system checking:
This is a type of testing, which involves the computer & the translator programs.

After entering the program, it is checked using a translator to detect any syntax errors. The translator can be a Compiler or an Interpreter, which goes through the set of instructions & produces a list of errors, or a program/statement listing which is free from errors.

Functional testing (White-box testing):
This type of testing is based upon examining the internal structure of a program & selecting test data, which give rise to the alternative cases of control flow.
Use of Test data.
The accuracy of a program can be tested by inputting a set of values referred to as **Test data**. The test data is designed to produce predictable output.

There are 2 types of test data;

(a). **Real data (live data)**: - test data obtained from the real problem environment (practical applications).

(b). **Dummy data**: - assumed test data.

The programmer invents simple test data, which he/she uses to carry out trial runs of the new program. At each run, the programmer enters various data variations including data with errors to test how the system will behave. For example, if the input required is of numeric type, the programmer may enter alphabetic characters. The programmer will then compare the output produced with the predicted (actual) output.

**Notes.**
- Where possible, the program should be tested using the same test data that was used for desk checking. More strict/rigid tests should be applied on the program in order to test the program to its limits.
- Only **Logical errors & Semantic errors** can be corrected by the programmer using test data.
- A good program should not crash due to incorrect data entry but should inform the user about the irregularity and request for the correct data to be entered.

Use of debugging utilities.
After the program has been entered in the program editor, debugging utilities which are built in the computer can be run during translation to detect any syntax errors in the program. The errors are corrected and the debugging process is repeated again to find out more errors, before the program is executed.

Diagnostic procedures.
For complex programs, diagnostic procedures, such as **Trace routines**, may be used to find logical errors. A Trace prints out the results at each processing step to enable errors to be detected quickly.

System Test with actual data.
This is whereby the new program is run in parallel with the existing system for a short time so that results can be compared and adjustments made. In such cases, the system test is made using actual data.

Review Questions.
1. Differentiate between Testing and Debugging.
2. What is Dry running?
Implementation and Maintenance.

IMPLEMENTATION

Implementation refers to the actual delivery, installation and putting of the new program into use.

The program is put into use after it is fully tested, well documented, and after training the staff who will be involved in the running of the new program.

Structured Walk Through:

It is an organized style of evaluating/reviewing a program by a team of other programmers, which then reports to the programming team.

REVIEW AND MAINTENANCE.

Once the program becomes operational, it should be maintained throughout its life, i.e., new routines should be added, obsolete routines removed, & the existing routines adjusted so that the program may adapt to enhanced functional environments.

The main objective of maintenance is to keep the system functioning at an acceptable level.

Program maintenance mainly involves:

✓ Correcting errors that may be encountered after the program has been implemented or exposed to extensive use.
✓ Changing procedures.
✓ Hardware and software maintenance.
✓ Changing parameters and algorithms used to develop the original programs.
✓ Making any adjustments as new technology comes.

Note. Program maintenance runs parallel to the maintenance of the program documentation, i.e., any time maintenance is carried out on the program, the documentation should also be updated to convey the right image of the system.

Program documentation.

After writing, testing, and debugging a program, it must be documented. In other words, the programmer should describe all what he was doing during the program development stages.

Program documentation is the writing of supportive materials explaining how the program can be used by users, installed by operators, or modified by other programmers.

Note. All the program development activities (i.e., from the initial stage up to the complete program) should be documented/recorded in order to assist in the development of the program, future modification of the program, general maintenance, machine & software conversion at a later date, and program changeover.

Documentation can either be; Internal or External.

Internal documentation is the writing of non-executable lines (comments) in the source program that help other programmers to understand the code statements.

External documentation refers to reference materials such as user manuals printed as booklets.

Types of program documentation.

There are 3 target groups for any type of documentation:

1. User-oriented documentation.

   This enables the user to learn how to use the program as quickly as possible, and with little help from the program developer.
2. **Operator-oriented documentation:**
   
   This is meant for computer operators such as the technical staff. It is used to help them install & maintain the program.

3. **Programmer-oriented documentation:**
   
   This is a detailed documentation written for skilled programmers. It provides the necessary technical information to help in future modification of the program.

**Some documents used in program documentation.**

1. **User guide/manual.**
   
   This is a manual provided for an end-user to enable him/her use or operate the program with minimal or no guidance.
   
   A User guide is used in user-oriented documentation.

2. **Reference guide.**
   
   It is used by someone who already knows how to use the program but needs to be reminded about a particular point or obtain more detailed information about a particular feature.

3. **Quick Reference guide.**
   
   This could be a single sheet or card small enough to fit into a pocket. It is used by the user to get help for the common tasks carried out within the program.

4. **Technical manuals.**
   
   They are intended for System analysts & Programmers. They assist in maintaining & modifying the program design and code.

**Contents in a program document.**

Documentation includes:

1. Title of the program.
2. Function of the program.
3. Language used.
4. Hardware & Software required to support the processing of the system.
5. File specifications (details of the data structures used, & details of how data files are to be organized, accessed, and kept secure).
6. Limitations of the program.
7. Format of the input & the output expected.
8. Design of the program using the design tools (i.e., detailed algorithms & procedures used).
9. A listing of the Source program and the program flowcharts.
10. A carefully devised set of Test data, and a table of expected results.
11. Detailed instructions on how to run the program.

**Review Questions.**

1. What is program designing?
2. (a) Define program documentation. 
   (b) What does a program documentation contain?
3. Briefly explain how each of the following documents are useful in programming?
   (a) User manual / guide.
   (b) Reference guide.
   (c) Quick reference guide.
4. Program documentation is different from Implementation. Explain.
5. Outline and briefly explain the stages involved in program development.
DEVELOPING OF ALGORITHMS

After carefully analyzing the requirements specification, the programmer usually comes up with the algorithm.

Definition of an Algorithm:

- An Algorithm is a limited number of logical steps that a program follows in order to solve a problem.
- A step-by-step (a set of) instructions which when followed will produce a solution to a given problem.
- Algorithms take little or no account of the programming language.
- They must be precise/accurate, unambiguous/clear and should guarantee a solution.

Program design Tools.

Algorithms can be illustrated using the following tools:
1). Pseudocodes.
2). Flowcharts.
3). Decision Tables.
4). Decision Trees.

Note. For any given problem, the programmer must choose which algorithm (method) is best suited to solve it.

PSEUODECODES.

- A pseudocode is a method of documenting a program logic in which English-like statements are used to describe the processing steps.
- These are structured English-like phrases that indicate the program steps to be followed to solve a given problem.
- The term “Code” usually refers to a computer program. This implies that, some of the words used in a pseudocode may be drawn from a certain programming language and then mixed with English to form structured statements that are easily understood by non-programmers, and also make a lot of sense to programmers. However, pseudocodes are not executable by a computer.

Guidelines for designing a good pseudocode.

1. The statements must be short, clear and readable.
2. The statements must not have more than one meaning (i.e., should not be ambiguous).
3. The pseudocode lines should be clearly outlined and indented.
4. A pseudocode must have a Begin and an end. i.e., a pseudocode should show clearly the start and stop of executable statements and the control structures.
5. The input, output and processing statements should be clearly stated using keywords such as PRINT, READ, INPUT, etc.
Example 1:
Write a pseudocode that can be used to prompt the user to enter two numbers, calculate the sum and average of the two numbers and then display the output on the screen.

START
    PRINT “Enter two numbers”
    INPUT X, Y
    Sum = X + Y
    Average = Sum/2
    PRINT Sum
    PRINT Average
STOP

Example 2:
Write a structured algorithm that would prompt the user to enter the Length and Width of a rectangle, calculate the Area and Perimeter, then display the result.

Solution
Step 1: Draw the rectangle of Length (L) and Width (W).
Step 2: Write down the Pseudocode.

START
    PRINT “Enter Length and Width”
    READ L, W
    Area = L * W
    Perimeter = 2 (L + W)
    PRINT Area
    PRINT Perimeter
STOP

Example 3:
Write a pseudocode that can be used to calculate the Diameter, Circumference and Area of a circle and then display the output on the screen.

START
    Set π to 3.14
    Prompt the user for the Radius (R)
    Store the radius in a variable (R)
    Set Diameter to 2 * Radius
    Set Circumference to π * 2 * Radius
    Set Area to π * Sqr (Radius)
    PRINT Diameter
    PRINT Circumference
    PRINT Area
STOP
Example 4:

Write a pseudocode for a program that would be used to solve equation: \( E = MC^2 \).

START
Enter values from M to C
\( E = M \times C \times C \)
Display E

STOP
✓ It is important to use program control structures when writing Pseudocodes. The most common constructs are:

(i). **Looping (Repetition / Iteration)** – used where instructions are to be repeated under certain conditions.
(ii). **Selection** – used when choosing a specified group of instructions for execution. The group chosen depends on certain conditions being satisfied.

Example 5:

Write a pseudocode for a program that can be used to classify people according to age. If a person is more than 20 years; output “Adult” else output “Young person”.

START
PRINT “Enter the Age”
INPUT Age
IF Age > 20 THEN
    PRINT “Adult”
ELSE
    PRINT “Young person”

STOP

Note. Pseudocodes make an algorithm easier to understand. This is because; the algorithm can be read from top to bottom without the need for jumping backwards or forwards to follow the logic of the algorithm as in flowcharts.

FLOWCHARTS.

✓ A **Flowchart** is a diagrammatic or pictorial representation of a program’s algorithm.
✓ It is a chart that demonstrates the logical sequence of events that must be performed to solve a problem.

Types of Flowcharts.

There are 2 common types of Flowcharts:

1). **System flowchart**.
   A System flowchart is a graphical model that illustrates each basic step of a data processing system.
   It illustrates (in summary) the sequence of events in a system, showing the department or function responsible for each event.

2). **Program flowchart**.
   This is a diagram that describes, in sequence, all the operations required to process data in a computer program.
   A program flowchart graphically represents the types of instructions contained in a computer program as well as their sequence & logic.
PROGRAM FLOWCHARTS.

A Flowchart is constructed using a set of special shapes (or symbols) that have specific meaning. Symbols are used to represent operations, or data flow on a flowchart. Each symbol contains information (short text) that describes what must be done at that point. The symbols are joined by arrows to obtain a complete Flowchart. The arrows show the order in which the instruction must be executed.

SYMBOLS USED IN PROGRAM FLOWCHARTS.

Below is a standard set of symbols used to draw program flowcharts as created by American National Standard Institute (ANSI).

1. Terminal symbol.  
   *Ellipse (Oval in shape)*
   
   It is used to indicate the point at which a flowchart, a process or an algorithm begins & ends.
   
   ✓ All Flowcharts must have a START & STOP symbol. The START/BEGIN symbol is the first symbol of a flowchart, & identifies the point at which the analysis of the flowchart should begin. The STOP/END symbol is the last symbol of a flowchart, & indicates the end of the flowchart.
   
   ✓ The words Begin & End (or Start & Stop) should be inserted in the Terminal symbol.

2. Input or Output symbol.
   *Parallelogram*
   
   - It is used to identify/specify an input operation or output operation.
   
   For example:
   
   ```plaintext
   READ Employee Name
   PRINT Employee Name
   
   Input operation   Output operation
   
   Note. The words mostly associated with I/O operations are READ & PRINT. READ describes the entry of computer data, while PRINT relates to the printed output of information.
   
      *Rectangle*
      
      - Process symbol is used to indicate that a processing or data transformation is taking place.
      
      The information placed within the process symbol may be an algebraic formula or a sentence to describe processing.
      
      ```plaintext
      SUM = A + B
      Commission is computed at 20% of Total Sales
      
      Processing defined as a Formula   Processing defined as a Sentence
      
      4. Decision symbol.
         *Rhombus*
         
         - It is used to indicate/ specify a condition or to show the decision to be made.
There are 2 main components of a Decision symbol:

(i). A question asked within the Decision symbol, that indicates the comparison / logical operation.

(ii). The results of the comparison (which are given in terms of YES or NO).

The arrows labeled YES or NO lead to the required action corresponding to the answer to the question.

5. **Flow lines.**

Flow lines with arrowheads are used to indicate the direction of processing of the program logic, i.e., they show the order in which the instructions are to be executed.

The normal flow of a flowchart is from *Top to Bottom*, and *Left to Right*.

**Note.** Flow lines should never cross each other.

6. **Connector symbol.**

Sometimes, a flowchart becomes too long to fit in a single page, such that the flow lines start crisscrossing at many places causing confusion & also making the flowchart difficult to understand.

The **Connector symbol** is used as a connecting point for arrows coming from different directions.

A Connector symbol is represented by a Circle, and a letter or digit is placed within the circle to indicate the link.

**Note.** Connectors do not represent any operation. They are used to connect two parts of a flowchart, indicating that the flow of data is not broken.

**General guidelines for drawing a program flowchart.**

1. A flowchart should have only one entry/starting point and one exit point (i.e., ensure that the flowchart has a logical start and finish).

2. The flowchart should be clear, neat and easy to follow.

3. Use the correct symbol at each stage in the flowchart.

4. The flowchart should not be open to more than one interpretation.

5. Avoid overlapping the lines used to show the flow of logic as this can create confusion in the flowchart.

6. Make comparison instructions simple, i.e., capable of YES/NO answers.

7. The logical flow should be clearly shown using arrows.
   
   **Note.** A flowchart should flow from the Top to Bottom of a page, and from the Left to the Right.

8. Where necessary, use Connectors to reduce the number of flow lines.
   
   Connectors are helpful when a flowchart is several pages long, and where several loops are needed in the logic of the flowchart.

9. Check to ensure that the flowchart is logically correct & complete.
Example 1:
Draw a flowchart for a program that can be used to prompt the user to enter two numbers, find the sum and average of the two numbers and then display the output on the screen.

```
Start
↓
X, Y
↓
Sum = X + Y
Average = Sum/2
↓
PRINT Sum, Average
↓
Stop
```

Example 2:
Draw a flowchart for a program that would prompt the user to enter the Length and Width of a rectangle, calculate the Area and Perimeter, then display the result.

```
Start
↓
L, W
↓
Area = L x W
Perimeter = 2 (L x W)
↓
PRINT Area, Perimeter
↓
Stop
```
Example 3:

Draw a flowchart for a program that can be used to calculate the Diameter, Circumference and Area of a circle and then display the output on the screen.

\[ \text{Diameter} = 2 \times \text{Radius} \]
\[ \text{Circumference} = \pi \times 2 \times \text{Radius} \]
\[ \text{Area} = \pi \times \text{Sqr} \text{(Radius)} \]

Example 4:

Design a flowchart for a program that can be used to classify people according to age. If a person is more than 20 years, output “Adult” else output “Young person”.

\[ \text{Age} > 20? \]

- Yes → Adult
- No → Young person
Example 5:

Draw a flowchart for a program that would be used to classify animals according to sex. If a letter M is input, the program should display ‘Male’ otherwise it should display “Female”.

Example 6:

Write a program using a flowchart to convert temperature from °C to °F. Fahrenheit = 32 + (9/5 * °C).

Notes.

- A flowchart must have a Start and an end.
- A flowchart is useful when the algorithm is short & the flowchart can fit conveniently on a single page. If the flowchart is too large, it is recommended to use Pseudocodes for long & complicated programs.
Advantages of using Flowcharts.

The following are the benefits of Flowcharts:

(i). **Quicker understanding of relationships.**

They assist programmers to understand procedures more quickly. A programmer can represent a lengthy procedure more easily with the help of a flowchart than describing it by means of written notes.

(ii). **Effective synthesis.**

Flowcharts may be used as working models in the design of new programs and systems.

(iii). **Proper program documentation.**

Program flowcharts serve as good program documentation, which is needed for the following reasons:

- **(a).** If programs are modified in future, the flowcharts will direct the programmer on what was originally done.
- **(b).** When staff changes occur, the flowcharts may help new employees understand the existing programs.
- **(c).** Flowcharts assist in program conversion when new hardware/software are acquired.

(iv). **Effective coding.**

Program flowcharts act as a guide during the program preparation stage. Instructions coded in a programming language may be checked against the flowchart to ensure that no steps are omitted.

(v). **Orderly debugging and testing of programs.**

Flowcharts help in detecting, locating and removing mistakes. The programmer can refer to the flowchart as he/she re-checks the coding steps, & the logic of the written instructions.

(vi). **Efficient program maintenance.**

Flowcharts facilitate the maintenance of operating programs. They help the programmer to concentrate on the part of the information flow which is to be modified.

Limitations of using Flowcharts.

(i). Flowcharts are complex, clumsy & become unclear, especially when the program logic is complex.

(ii). If changes are to be made, the flowchart may require complete re-drawing.

(iii). Reproduction of flowcharts is usually a problem, since the flowchart symbols cannot be typed.

(iv). No uniform practice is followed for drawing flowcharts as it is used as an aid to the program.

(v). Sometimes, it becomes difficult to establish the link between various conditions, and the actions to be taken upon a particular condition.

Revision Exercise.

1. Define the following:
   - (a). Algorithm.
   - (b). Pseudocode.
   - (c). Flowchart.

2. (a). State the various types of flowcharts.
   - (b). Discuss the advantages and disadvantages of flowcharts.
PROBLEM SOLVING

1. Problem Identification (problem recognition).

Write a program which:
(a). Requests the user to enter a temperature in °C.
(b). Calculates the corresponding temperature in °F.
(c). Outputs the given temperature and the converted value.

2. Problem definition & Problem Analysis.

- Determine the general requirements, i.e., the main inputs to the program, the main outputs from the program, & also the kind of files which may be needed.
Find out how to convert the given temperature. If the given temperature is in °C, then convert it to °F.
\[ F = 32 + \left( \frac{9}{5} \right) \times °C \]
- The Keyboard will be used to enter the Centigrade temperature, and display the output on the Screen.

3. Design the program.

Develop an Algorithm (a method) for solving the problem.

An Algorithm is a set of instructions which when followed will produce a solution to a given problem.
- Write the instructions in such a way that they can be easily converted into a form which the computer can follow.

Computer instructions fall into 3 main categories:
(a). Input instructions – used for supplying data to a program inside the computer. The data supplied is stored in the memory of the computer.
(b). Processing instructions – used for manipulating data inside the computer.
These instructions allow us to Add, Subtract, Multiply, & Divide. They also allow us to compare two values, and act according to the result of the comparison.
(c). Output instructions – used to get information out of the computer.

Note. The programmer must choose which algorithm (method) is the best suited to solve it. This may involve drawing a Flowchart or writing Pseudocode.

Algorithm.

STEP 1: [Prompt the user to enter temperature in °C ]
STEP 2: [Store the value in memory]
STEP 3: [Calculate the corresponding temperature in °F]
STEP 4: [Store the result in memory location]
STEP 5: [Output the values in °C & °F]
STEP 6: [Stop]

After the algorithm is developed, it must be checked by use of appropriate data values to make sure it is doing its job correctly. This process is called Dry running or Desk checking the algorithm, & is used to pin-point any errors in logic before the program is actually written.

Note. You should never start writing programming codes unless you are absolutely sure that the algorithm is correct.
- Algorithms do not depend on any particular language.

**Flowchart.**

Flowcharts are useful for specifying small algorithms.
A flowchart consists of a set of ‘flowchart symbols’ connected by arrows. Each symbol contains information about what must be done at that point & the arrows show the order in which the instructions must be executed.

4. **Write a computer program corresponding to the algorithm.**
   - Choose a suitable language & write the computer program using the algorithm.
   - Use the appropriate language statements & control structures which are found in that programming language.

5. **Test the program to find out whether it is doing its intended job.**
   - Testing involves supplying data values (Test data) to the program for which the answer is known. Some values we can use are 0, 10, 100.
   - If the program does not give us the correct answers, then we know the program contains some errors.

6. **Debug the program.**
   - I.e., find & correct any errors in the program.

7. **Document the program.**
   Write out an explanation of how the program works, and how to use it. This includes:
   (i). The statement of the problem.
   (ii). The Algorithm for solving the problem.
   (iii). The program listing.
   (iv). Test data, and the results produced by the program.

**Note.** Documentation should be done at the same time as all the other activities. Each activity produces its own items of documentation which will be part of the final program documentation.
1). **User documentation (User guide)**

It enables a non-technical person to use the program without the need to know about the internal workings of the program.

**User guides** are intended to help the user to use the program (to operate it) with minimal or no guidance.

**Example 1:**

Assume that the program is called **TEMPCONV**, and resides on a disk named **CONVERT**.

**Program Name:** TempConv

**Machine:**

This program is designed for use on IBM Microcomputers.

**Purpose:**

This program simulates the conversion of temperature in °C to the corresponding temp. in °F. *(Attach a description of how the game is played)*.

The strategy used is for the program to request the user to enter a temperature in °C. The user specifies this temperature, and the program calculates the corresponding temperature in °F, printing the given temperature and the converted value.

**Location:**

On the disk named **CONVERT**

**How to Use:**

Start PASCAL in drive C:
Insert the disk **CONVERT** into the disk drive.
Press ALT + ‘F’ to open the File menu, then choose Open.
Type A:TEMPCONV, then press ENTER

**Input:**

The program requests the user to enter a temp. in °C. When the message “ENTER TEMP. IN DEGREES CENTIGRADE” appears on the screen, you must type the value of the temperature you want converted.

Only digits 0 to 9, a possible sign, and a possible decimal point must be typed. E.g., 25, -20, or 30.5. Typing 25C, for instance, is invalid and will result in an error.

**Output:**

The program will print a heading followed by the two temperatures. For example, if 40 is entered as the Centigrade temp., the output will be;

<table>
<thead>
<tr>
<th>Centigrade</th>
<th>Fahrenheit</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>104</td>
</tr>
</tbody>
</table>

**Example 2:**

**Program Name:** OneZero

**Machine:**

This program is designed for use on an IBM Microcomputer.
Purpose:
This program simulates the playing of the One zero game. (A *description of how the game is played should be attached*).
The strategy used is to limit a turn to a maximum number of throws. The user specifies this maximum (as well as the number of turns to play), and the program simulates a game, printing the score obtained.
The program can be used to enable the user to select the maximum number of throws to make on each turn, in order to increase the chances of winning an actual game.

Location:
On the disk named GAMES

How to Use:
Start PASCAL in your computer.
Insert the disk GAMES into the disk drive.
Choose Open from the File menu.
Type A:OneZero, then press ENTER
Press ALT + ‘R’

When the prompt: ‘ENTER NUMBER OF TURNS PER GAME’ appears, type the number of turns for which you want a game to last. Thus, if you want the game to consist of 20 turns, type 20.

When the prompt: ‘ENTER MAXIMUM THROWS PER TURN’ appears, type a number, say, 5.
The program will simulate one game of 20 turns; each turn will consist of a maximum of 5 throws. When it is finished, it will print the results as in the following sample:

*The game consisted of 20 turns*
*Each turn consisted of a maximum of 5 throws*
*The score obtained for the game was 156*

It will then display the prompt: ENTER MAXIMUM THROWS PER TURN.
Again, you can enter another number (or the same one, if you wish), and the computer will simulate another game of 20 turns, using the new maximum that you have given. If you don’t wish to continue, you can enter 0 at this stage and the program will end.

Restrictions:
If you wish to vary the number of turns in a game, the program has to be rerun. You can type: ALT + ‘R’, and enter the required number when the prompt: ‘ENTER NUMBER OF TURNS PER GAME’ appears.
2. **Technical documentation (Technical manuals)**

Technical manuals are intended for the systems analysts or programmers. This documentation is useful to a programmer & can help them in maintenance & modification of the program design and code at a later stage.

**Example**

**Program Name:** DiceGame

**Purpose:**

This program simulates the playing of a dice game. *(Attach a description of how the game is played).*

The strategy used is to limit a turn to a maximum number of throws. The user specifies this maximum (as well as the number of turns to play), and the program simulates a game, printing the score obtained.

**Program Structure:**

The program consists of three modules;

- Main
- SimulateOneGame
- SimulateOneTurn

The following diagram shows the relationship between these modules:

```
  MAIN
   |
   v
SimulateOneGame
   |
   v
SimulateOneTurn
```

(Note. Here give the documentation for each individual module. The documentation given is only for the module SimulateOneTurn. Therefore, develop the documentation for the other modules along similar lines).

**Module Name:** SimulateOneTurn

**Parameters:** MaxThrowsPerTurn, ScoreThisTurn

**Purpose:**

Given MaxThrowsPerTurn, this module simulates one turn and returns (in ScoreThisTurn), the score obtained for that turn.

**Variables used:**

- `MaxThrowsPerTurn` - a parameter representing the maximum number of throws per turn. This value is supplied to the module.
- `ScoreThisTurn` - a parameter used to return the score for the turn to the calling module.
- `NumberOfThrows` - used to count the number of throws made. If the count reaches the maximum, the turn ends.)
Modules Called:
Only the standard module RANDOM is called to simulate the throwing of the dice. RANDOM (1, 6) produces a random number in the range 1 to 6, inclusive.

The Algorithm:
MODULE SimulateOneTurn (MaxThrowsPerTurn, ScoreThisTurn)
Set ScoreThisTurn to 0
Set NumberOfThrows to 0
WHILE NumberOfThrows < MaxThrowsPerTurn DO
Set ThrowValue to RANDOM (1, 6)
Add 1 to NumberOfThrows
IF ThrowValue = 1 THEN
Set ScoreThisValue to 0
Set NumberOfThrows to MaxThrowsPerTurn {force loop exit}
ELSE
Add ThrowValue to ScoreThisTurn
ENDIF
ENDWHILE
ENDMODULE

Explanation Notes:
If a 1 is thrown, the turn ends. In this case, a forced exit of the WHILE loop is made. This is done by setting NumberOfThrows to MaxThrowsPerTurn.

Note. If the module was tested individually; a program listing, the test data used, and the results obtained should be included in the documentation of the module. After each module has been documented, sample runs of the entire program should be added. This should include:
• The complete program listing.
• Test data used.
• Results obtained.

Review Exercise
1. Why is documentation an essential part of the program development process?
2. Name four items which the user documentation of a program must contain.
3. What items make up the technical documentation of a program?
PROGRAM CONTROL STRUCTURES

Control structures are blocks of statements that determine how program statements are to be executed.

Control statements deal with situations where processes are to be repeated several number of times or where decisions have to be made.

There are 3 control structures used in most of the structured programming languages:

1). Sequence.
2). Selection.
3). Iteration (looping).

SEQUENCE CONTROL STRUCTURES

In Sequence control, the computer reads instructions from a program file line-by-line starting from the first line sequentially towards the end of the file. This is called Sequential program execution.

Start → Statement 1 → Statement 2 → … → Statement n → End

Note. Sequential program execution enables the computer to perform tasks that are arranged consecutively one after another in the code.

SELECTION (DECISION) CONTROL STRUCTURES

Selection involves choosing a specified group of instructions/statements for execution.

In Selection control, one or more statements are usually selected for execution depending on whether the condition given is True or False.

The condition must be a Boolean (logical) expression, e.g., X >= 20
In this case, the condition is true if x is equal to or greater than 20. Any value that is less than 20, will make the condition false.

Generally, there are 4 types of selection control structures used in most high-level programming languages:

1). IF – THEN
2). IF – THEN – ELSE
3). Nested IF
4). CASE – OF

Note. These control structures are used in a program based on a sequence of instructions, which require that a choice (decision) be made between two or more alternatives.
In such a situation, the computer must be programmed to compare data, and take action depending on the outcome of the comparison.

IF – THEN

IF – THEN structure is used if only one option is available, i.e., it is used to perform a certain action if the condition is true, but does nothing if the condition is false.

The general format of the IF-THEN structure is:

IF < Condition > THEN
  Program statement to be executed if condition is true;
ENDIF
If the condition is TRUE, the program executes the part following the keyword ‘THEN’. If the condition is FALSE, the statement part of the structure is ignored, and the program continues with the statements below the ENDIF.

The diagrammatic expression of the IF-THEN structure is:

![IF-THEN Diagram]

Example 1:
In a school, the administration may decide to reward only those students who attain a mean mark of 80% and above.

**Pseudocode**

IF Mark > 80 THEN

    Print “Give reward”

ENDIF

Example 2:
A user is asked to enter a set of positive numbers, one at a time. She enters a 0 (zero) to indicate that she has no more numbers to enter. Develop an algorithm to print the largest number entered.

**Pseudocode**

START

Prompt the user for a number, Largest
Prompt the user for another number, NewNumber

    IF NewNumber > Largest THEN
        Set Largest to NewNumber
    ENDIF

Prompt the user for a number, NewNumber
Output (‘The largest number entered is’, Largest)

STOP
As each number is entered, the algorithm checks if the number entered is larger than the previous ones. If it is larger, it is saved as the largest. If it is smaller, it is ignored, and holds onto the largest number so far.

**Example 3:**

```plaintext
PROGRAM AgeTalk (Input, Output);
VAR Age: INTEGER;
BEGIN {program}
  Writeln (‘How old are you?’); Readln (Age);
  IF Age >= 18 THEN
    Writeln (‘You are old enough to join the army.’);
END. {program}
```

**Note.** Compound statements can also be used with the IF – THEN structure.

**Example 4:**

```plaintext
PROGRAM Service; {*This program displays a message depending on the number of years you have worked for a company*}
VAR Years: INTEGER;
BEGIN
  CLRSCR
  Writeln (‘How long have you been with the company?’); Readln (Years);
  IF Years > 20 THEN
    Writeln (‘Get a Gold watch’);
  IF (Years > 10) AND (Years <= 20) THEN
    Writeln (‘Get a Paper weight’);
  IF Years <= 10 THEN
    Writeln (‘Get a pat on the back ’);
END.
```

**IF - THEN - ELSE**
The IF-THEN-ELSE structure is suitable when there are 2 available options to select from.
The general format of the IF-THEN-ELSE structure is:

```
IF < Condition > THEN
  Statement 1; (called the THEN part)
ELSE
  Statement 2; (called the ELSE part)
ENDIF (indicates the end of the control structure)
```
The diagrammatic expression of the IF-THEN-ELSE structure is:

```
Condition?  TRUE  FALSE

<table>
<thead>
<tr>
<th>Execute statements between THEN &amp; ELSE</th>
<th>Execute statements between ELSE &amp; ENDIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute statements after ENDIF</td>
<td></td>
</tr>
</tbody>
</table>
```

Continuation of program

When the IF-THEN-ELSE structure is encountered:

1. The Condition is tested.
2. If the Condition is TRUE, the statements between THEN & ELSE (i.e., the THEN part) are executed. The ELSE part is skipped, and execution continues with the statement following ENDIF.
3. If the Condition is FALSE, the THEN part is skipped. The statements between ELSE & ENDIF (i.e., the ELSE part of the structure) are executed, and execution continues with the statement following ENDIF.

After either group of statements has been executed, the program will then continue executing the program statements after the last ENDIF.

**Note.** Using IF-THEN-ELSE, for any given test of the condition, only one set of statements is selected for execution (not both statements).

**Example 1:**

In a football match, if a player makes a mistake which is considered serious by the rules of the game, he/she is given a Red card. Otherwise, he/she is given a Yellow card.

**Pseudocode**

```
IF Fault = Serious THEN
  Print “Give red card”
ELSE
  Print “Give Yellow card”
ENDIF
```

**Example 2:**

Write an algorithm which asks a user for two numbers; A and B, and calculates the value of A divided by B. However, if B is 0, a message is printed which says that division by 0 is not allowed.
Pseudocode

START

Prompt the user for the two numbers, A and B

IF B = 0 THEN
    Writeln (‘Division by 0 is not allowed’) ELSE
    Set C to A/B
ENDIF

Output A, B, and C

STOP

Explanation.

♦ Suppose the user enters 1 for A and 0 for B in response to the prompt. The algorithm will test if B=0. Since B is 0, the condition is True. Therefore, the THEN part is executed printing the message: ‘Division by 0 is not allowed’.

♦ Suppose the user enters 20 for A and 5 for B in response to the prompt. The algorithm will test if B=0. Since B is not 0, the condition is False. Therefore, the statements between ELSE & ENDIF are executed (i.e., A is divided by B, and the result is stored in C).

NESTED IF

Nested IF structure is used where 2 or more options have to be considered to make a selection. The general format of the Nested IF structure is:

IF < Condition 1 > THEN
    Statement 1
ELSE
    IF < Condition 2 > THEN
        Statement 2
    ELSE
        IF < Condition 3 > THEN
            Statement 3
        ELSE
            Statement 4;
        ENDIF
    ENDIF
ENDIF

Example:

In an Olympics track event, medals are awarded only to the first three athletes as follows:

a). Position 1: Gold medal
b). Position 2: Silver medal
c). Position 3: Bronze medal

The pseudocode and flowchart below can be used to show the structure of the Nested IF selection.
**Pseudocode**

IF Position = 1 THEN
   Medal = “Gold”
ELSE
   IF Position = 2 THEN
      Medal = “Silver”
   ELSE
      IF Position = 3 THEN
         Medal = “Bronze”
      ELSE
         Medal = “nil”
      ENDIF
   ENDIF
ENDIF

**Flowchart**

When IF statements are embedded within one another, they are said to be **Nested**.

**Note.** Each IF-THEN or IF-THEN-ELSE is terminated with the comment {ENDIF}. The number of {End If’s} must be equal to the number of ELSE’s.

**The CASE structure**

CASE-OF allows a particular group of statements to be chosen from several available groups. It is therefore used where the response to a question involves more than two choices/alternatives.

The general format of the CASE structure is:

CASE *Expression* OF
   
   *Label 1*: statement 1
   *Label 2*: statement 2
   *Label 3*: statement 3

---

- 46 -
The Boolean expression for the CASE structure can only be expressed using *Integers* or *alphabetic characters* only. Hence;  
CASE Integer OF or CASE Char OF

A statement is executed only if one of its corresponding labels matches the current value of the expression. This implies that, the current value of the expression determines which of the statements will be executed.

**Example 1:**

Write a pseudocode of a program that requests the user to type a number from 1 to 7. The program then prints the corresponding day of the week.

**Pseudocode**

START
   Prompt the user for a number from 1 to 7, Day
   CASE Day OF
      1: Writeln (‘Sunday’);
      2: Writeln (‘Monday’);
      3: Writeln (‘Tuesday’);
      4: Writeln (‘Wednesday’);
      5: Writeln (‘Thursday’);
      6: Writeln (‘Friday’);
      7: Writeln (‘Saturday’);
   END CASE
STOP

The CASE structure consists of:

- The word CASE.
- A Control variable (e.g., Day).
- The word OF.
- A group of one or more statements, each group labeled by one or more possible values of the control variable.
- The word ENDCASE, indicating the end of the construct.

When a CASE statement is encountered, the value of the control variable is used to determine which group of statements is executed, e.g., if the value of Day is 5, then the group of statements labeled 5 is selected for execution, and the statement; ‘Thursday’ is printed.

After executing this group of statements, execution continues at the statement following ENDCASE.
NOTES:
i). The programmer should ensure that the value of the control variable appears as a label. E.g., suppose the value entered for Day was 9. Since 9 does not label any statement within the CASE construct, an error will result.

ii). A given label can be used on only one group of statements. E.g., 5 can’t be used to label two groups of statements. If this is done, the computer will not know which group to select & unpredictable results can occur.

Example 2:
Write a pseudocode of a program that requests the user to type a number from 1 to 7. Depending on the number entered, print the message, ‘It is a School day’ or ‘It is on a Weekend’.

Pseudocode
Prompt the user for a number from 1 to 7, Day

IF (Day < 1) OR (Day > 7) THEN
    Print (‘Invalid number entered ---’, Day)
ELSE
    CASE Day OF
        2, 3, 4, 5, 6: Writeln (‘It is a School day’);
        1, 7: Writeln (‘It is on a Weekend’);
    ENDCASE
ENDIF
STOP

In this pseudocode, the IF statement has been used to validate the value of Day. This ensures that, only valid data gets processed by the CASE statement. Otherwise, if the ELSE part is executed, we are sure that the value of Day will lie between 1 and 7 inclusive.

Example 3:

Pseudocode
CASE Average OF
    80 .. 100: Grade = ‘A’
    70 .. 79: Grade = ‘B’
    60 .. 69: Grade = ‘C’
    50 .. 59: Grade = ‘D’
    40 .. 49: Grade = ‘E’
ELSE
    Grade = ‘F’
ENDCASE
Flowchart

Example 4:
PROGRAM CaseSample (Input, Output);
VAR Grade:CHAR;
BEGIN {Program}
    Writeln (‘What grade did you get?’); Readln (Grade);
    CASE Grade OF
        ‘A’, ‘B’ : Writeln (‘Very Good’);
        ‘C’ : Writeln (‘Pass’);
        ‘D’, ‘F’ : Writeln (‘Wake up’);
    End; {Case}
    Readln;
End. {Program}

ITERATION (LOOPING / REPETITION) CONTROL STRUCTURES

Looping refers to the repeated execution of the same sequence of statements to process individual data. This is normally created by an unconditional branch back to a previous/earlier operation.

The loop is designed to execute the same group of statements repeatedly until a certain condition is satisfied.

Note. Iteration is important in situations where the same operation has to be carried out on a set of data many times.

The loop structure consists of 2 parts:
1). Loop body, which represents the statements to be repeated.
2). Loop control, which specifies the number of times the loop body is to be repeated.

Types of loops:
(a). Conditional loop: - This is where the required number of repetitions is not known in advance.
**Pseudocode**

**STEP 1:** [Prompt the user for temperature in °C]

**STEP 2:** [Store the value in memory]

**STEP 3:** IF C = 0 THEN Stop

**STEP 4:** [Calculate temperature in °F]

\[
F = 32 + \left(\frac{9}{5} \times °C\right)
\]

**STEP 5:** [Output temperature in °C & °F]

**STEP 6:** [GOTO Step 1]

**Flowchart**

![Flowchart diagram](image)

This algorithm illustrates **Conditional execution**. Conditional execution is a situation that requires that a logical test be carried out, and then a particular action be taken depending on the outcome of that test.

In this case, going to Step 4 will depend on whether the condition is True or False. E.g., If C = 10 then the condition ‘C = 0’ is False, and the program goes to Step 4. But if C = 0, then the condition is True, and the program stops.

(b). **Unconditional loop:** - This is where the execution of the instructions is repeated some specified number of times.

(c). **Continuous (infinite/unending) loop:** - This is where the computer repeats a process again and again, without ending.

**Example:**

**STEP 1:** [Prompt the user for temperature in °C]

**STEP 2:** [Store the value in memory]
STEP 3: [Calculate temperature in °F]
   \[ F: = 32 + \left( {°C \times \frac{9}{5}} \right) \]

STEP 4: [Output temperature in °C & °F]

STEP 5: [GOTO Step 1]

As long as a number is entered for °C, the algorithm does not stop when it reaches STEP 5 but rather transfers control to STEP 1, causing the algorithm/process to be repeated. However, a zero (0) can be used to stop the program because; the program cannot give the Fahrenheit equivalent to 0 °C.

Requirements for loops:
1. Control variable (Counter): - it tells/instructs the program to execute a set of statements a number of times.
2. Initialization: - allocating memory space, which will be occupied by the output.
3. Incrementing: - increasing the control variable by a certain number before the next loop.

Generally, there are 3 main looping controls:
1. The WHILE loop
2. The REPEAT…UNTIL loop.
3. The FOR loop.

The FOR loop
The FOR loop is used in situations where execution of the chosen statements has to be repeated a predetermined number of times.

The general format of the FOR loop is:

\[
\text{FOR loop variable } = \text{Lower limit TO Upper limit DO} \\
\text{Statements;} \\
\text{END FOR}
\]

Example:
Consider a program that can be used to calculate the sum of ten numbers provided by the user. The ‘FOR’ loop can be used to prompt the user to enter the ten numbers at most 10 times. Once the numbers have been entered, the program calculates and displays the accumulated sum.
**Pseudocode**

FOR count = 1 TO 10 DO
    PRINT “Enter a number (N)”
    Sum = Sum + N
END FOR

Display SUM

**Flowchart**

- Lower limit = Count + 1
- Count = Lower limit
- Sum = Sum + N
- Count <= 10?
  - YES
  - NO
    - Sum

**Explanation**

1. The *loop variable* (Count) is first initialized/set to the *Lower limit* whose value is 1.
2. The lower limit is then tested against the *Upper limit* whose value is set at 10.
3. If the lower limit is less than or equal to 10, the program will prompt the user to enter a number N, otherwise the computer will exit the loop.
4. After the last statement in the loop has been executed, the loop variable (count) is incremented by a 1 and stored in the lower limit, i.e., *Lower limit = Count + 1*.
5. The lower limit is again tested, and if it is less than or equal to 10, the loop is repeated until the time the lower limit will equal the upper limit.

**NOTE:**

The FOR loop can also be used to count downwards from the upper limit to the lower limit.

E.g., FOR count = 10 DOWN TO 1 DO

In this case, the upper limit 10 is tested against the lower limit 1.

**Pseudocode for a ‘FOR’ loop that counts from upper limit down to the lower limit:**

FOR *loop variable* = *Upper limit* DOWN TO *Lower limit* DO
    Statements;
END FOR

The flowchart extract for a FOR loop that counts downwards is:

- Loop variable = Upper limit
- Upper limit = Loop variable - 1
- Statements
- Condition?
  - NO
  - YES
The WHILE loop

The ‘WHILE’ loop is used if a condition has to be met before the statements within the loop are executed.
E.g., to withdrawal money using an ATM, a customer must have a balance in his/her account. Therefore, it allows the statements to be executed zero or many times.

**Pseudocode**

WHILE Balance > 0 DO  
    Withdraw cash  
    Update account  
ENDWHILE

**Explanation**

1. The condition balance > 0 is first tested.  
2. If it is TRUE, the account holder is allowed to withdraw cash.  
3. The program exits the loop once the balance falls to zero.

The general representation of the WHILE loop is:

**Pseudocode segment**

WHILE Condition DO  
    Statements;  
ENDWHILE

**Flowchart extract**

The REPEAT...UNTIL loop

In REPEAT...UNTIL, the condition is tested at the end of the loop. Therefore, it allows statements within it to be executed at least once.

E.g., if REPEAT...UNTIL is used in case of the ATM cash withdrawal, the customer will be able to withdraw the cash at least once since availability of balance is tested at the end of the loop.
Pseudocode
REPEAT
   Withdraw cash
   Update account
UNTIL balance <= 0;

The general format of the REPEAT…UNTIL loop is:

Pseudocode segment
REPEAT
   Statements;
UNTIL Condition;

Flowchart
\begin{figure}
\centering
\begin{tikzpicture}
    \node [rectangle, draw] (a) {Withdraw cash
        Update account};
    \node [diamond, below of=a] (b) {Balance > 0?};
    \node [below of=b] (c) {Exit loop};
    \node [right of=a] (d) {Yes};
    \node [left of=a] (e) {No};
    \draw [->] (a) -- (b);
    \draw [->] (b) -- (c);
    \draw [->] (b) -- (d);
    \draw [->] (b) -- (e);
\end{tikzpicture}
\end{figure}
DEVELOPING COMPLEX ALGORITHMS

Example 1:  
With aid of a pseudocode and a flowchart, design an algorithm that:

a). Prompt the user to enter two numbers X and Y.

b). Divide X by Y. However, if the value of Y is 0, the program should display an error message “Error: Division by zero”.

Pseudocode
START
PRINT “Enter two numbers X and Y”
INPUT X, Y
IF Y = 0 THEN
    PRINT “Error: Division by zero”
ELSE
    Quotient = X/Y
    PRINT X, Y, Quotient
ENDIF
STOP

Flowchart

Example 2:  
In an athletics competition, an athlete is rewarded as follows:

1st position: Gold
2nd position: Silver
3rd position: Bronze

Draw a pseudocode and a flowchart for a program that would be used to determine the type of medal to be rewarded to each athlete.
**Pseudocode**

START

PRINT “Enter athlete Name and Position”
INPUT Name, Position
IF Position = 1 THEN
  Medal = “Gold”
ELSE
  IF Position = 2 THEN
    Medal = “Silver”
  ELSE
    IF Position = 3 THEN
      Medal = “Bronze”
    ELSE
      Medal = “None”
    ENDIF
  ENDIF
ENDIF
ENDIF

**Flowchart**
Example 3:
The class teacher of Form 3S in a secondary school requested a programmer to design for her a simple program that would help her do the following:

(a) Enter the names of students and marks obtained in 8 subjects – Mathematics, English, Kiswahili, Biology, Chemistry, Business studies, Computer studies, and History.

(b) After entering the mark for each subject, the program should calculate the total and average marks for each student.

(c) Depending on the Average mark obtained, the program should assign grade as follows:

(i) Between 80 and 100 – A
(ii) Between 70 and 79 – B
(iii) Between 60 and 69 – C
(iv) Between 50 and 59 – D
(v) Below 50 – E

(d) The program should then display each student’s Name, Total marks and the Average grade.

Using a pseudocode and a flowchart, write an algorithm that shows the design of the program.

Pseudocode

START

REPEAT

PRINT “Enter student Name and subject marks”
INPUT Student name, Maths, Eng, Kisw, Bio, Chem, Business, Computer, History

SUM = Maths + Eng + Kisw + Bio + Chem + Business + Computer + History

AVG = SUM/8

IF (AVG => 80) AND (AVG <= 100) THEN

Grade = “A”

ELSE

IF (AVG => 70) AND (AVG <= 79) THEN

Grade = “B”

ELSE

IF (AVG => 60) AND (AVG <= 69) THEN

Grade = “C”

ELSE

IF (AVG => 50) AND (AVG <= 59) THEN

Grade = “D”

ELSE

Grade = “E”

ENDIF

ENDIF

ENDIF

PRINT Student name, Sum, AVG, Grade

UNTIL Count = Number of students

STOP
Example 4:
The gross salary of employees in ZAG BOOKS ENTERPRISE is based on basic salary and additional benefits as follows:

(a) Employees who have worked for the company for more than 10 years receive an additional pay of 10% to their basic salary.

(b) Monthly salary bonus based on monthly sales of books as follows:

<table>
<thead>
<tr>
<th>Monthly sales</th>
<th>Bonus Rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above 500,000</td>
<td>15</td>
</tr>
<tr>
<td>Between 250,000 and 500,000</td>
<td>10</td>
</tr>
<tr>
<td>Below 250,000</td>
<td>5</td>
</tr>
</tbody>
</table>

Draw a flowchart for a program that would be used to calculate the gross salary then output each employee’s basic salary, gross salary and all benefits.
Example 5:
A lady deposits 2,000 shillings in a Microfinance company at an interest rate of 20% per annum. At the end of each year, the interest earned is added to the deposit and the new amount becomes the deposit for that year.
Write a pseudocode for a program that would track the growth of the deposits over a period of seven years.
START

INPUT Initial Deposit
INPUT Interest Rate
SET Deposit to Initial deposit (i.e., 2000)

SET Year to 0

WHILE Year <= 7 DO
    Interest = Deposit x Interest rate
    Total = Deposit + Interest
    Deposit = Total  \{the new deposit\}
    Year = Year + 1
ENDWHILE

PRINT Deposit, Year
STOP

Example 6:

Draw a flowchart for a program that is to prompt for N numbers, accumulate the sum and then find the average. The output is the accumulated totals and the average.

Example 7:

Mutuku took a loan of Ksh. 400,000 from a local bank at an interest rate of 10% payable in four years. Assuming you wish to develop a computer program that will keep track of monthly repayments:

(a) Identify the input, processing and output requirements for such a program.
(b) Design the algorithm for the program using a simple flowchart and pseudocode.
(a). **Requirements:**

Input
- Initial amount borrowed
- Interest rate
- Number of years

Processing
- equation to calculate Yearly repayments and Monthly repayments.

Output
- Monthly repayments calculated by the process

(b). **Pseudocode:**

```
START

INPUT Initial amount borrowed
INPUT Interest rate
INPUT Number of years

Calculate Yearly repayments

Monthly repayments = (Yearly repayments / 12)

OUTPUT Monthly repayments

STOP
```

**Flowchart:**

1. **BEGIN**
2. ENTER Initial amount, Interest rate, number of Years
3. Calculate Yearly repayments & Monthly repayments
4. PRINT Monthly repayments
5. **END**