WHAT IS A COMPUTER PROGRAM?

A computer program is a series of coded instructions for the computer to obey and represent a method of processing data.

Programs can’t be written in English. They must first be written using a special language called a programming language. **A PROGRAMMING LANGUAGE** (e.g. BASIC, PASCAL, and C+) consists of a set of codes and rules which can be used to construct commands for the computer. These commands are read and translated into electronic pulses needed to make the computer work. Programs are written by programmers.

COMPUTER LANGUAGES

A computer language is a set of instructions used for writing computer programs. There are THREE (3) levels of languages:

1. **MACHINE LANGUAGE** – this was the first language available for programming. It varies from one computer to another, but the basic principles are the same. MACHINE LANGUAGE PROGRAMS are written using a series of 0’s and 1’s i.e. using a BINARY SYSTEM. All programs written today must be translated into machine language before they can be executed (used) by the computer.
   
   EXAMPLE: 110110001

2. **ASSEMBLY LANGUAGE / LOW LEVEL LANGUAGE** – these were developed to replace the 0’s and 1’s of machine language with symbols that are easier to understand and remember. Like with machine language, Assembly language varies form one make of computer to another so that a program written in one assembly language will not run on another make of computer.

   EXAMPLE: 
   LDA 300
   ADD 400
   STA 500

3. **HIGH LEVEL LANGUAGE** – these differ from low level languages in that they require less coding detail and make programs easier to write. High level languages are designed for the solution of problems in one ore more areas of the application and are commonly described as application-oriented or problem-oriented languages. High level languages are not machine dependant. Programs written in a high level language must be translated to a form which can be accepted by that computer, i.e.
they must be converted into machine code. This is achieved by means of a special program – either a **COMPILER** or an **INTERPRETER**.

**COMPILER**: compilers translate high-level programs into machine code. Each high level instruction normally converts into many machine instructions. The input (i.e. the high level code) into the compiler is called the source code and the output (i.e. the machine level code) is called the object code. The process of converting high level language to machine language in this way is called **COMPILING**.

**INTERPRETERS**: like the compiler, this is a special program which translates code written in a high level language into machine language. In contrast to a compiler, an interpreter does not translate the whole program prior to execution. It translates the code line by line during the execution of the program. This has the effect of slowing down the running of programs. With the compiler, all the translation is done first, and then the object program is executed.

---

**Diagram:**

```
HIGH LEVEL LANGUAGE
SOURCE CODE

COMPILER

MACHINE LANGUAGE
OBJECT CODE
```
## COMPARISON OF THE DIFFERENT LEVELS OF LANGUAGES

<table>
<thead>
<tr>
<th>LANGUAGE</th>
<th>ADVANTAGE</th>
<th>DISADVANTAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MACHINE LANGUAGE</strong></td>
<td>➢ Programs execute fast</td>
<td>➢ Programming slow and tedious.</td>
</tr>
<tr>
<td></td>
<td>➢ Code difficult to learn, read, understand and correct.</td>
<td>➢ Code is difficult to learn, read, understand and correct.</td>
</tr>
<tr>
<td></td>
<td>➢ Code is machine specific i.e. can’t be used in the same form on other computer models.</td>
<td></td>
</tr>
<tr>
<td><strong>ASSEMBLY LANGUAGE</strong></td>
<td>➢ Programming faster and less tedious than for machine language.</td>
<td>➢ Programming slower and more tedious than for high level languages.</td>
</tr>
<tr>
<td></td>
<td>➢ Code is easier to learn, read and understand than for machine language.</td>
<td>➢ Code is machine specific</td>
</tr>
<tr>
<td></td>
<td>➢ Execution faster than high level languages.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>➢ More compact than high level languages.</td>
<td></td>
</tr>
<tr>
<td><strong>HIGH LEVEL LANGUAGES</strong></td>
<td>➢ Programming faster and less tedious.</td>
<td>➢ Program executes slower.</td>
</tr>
<tr>
<td></td>
<td>➢ Code is easier to learn, read, understand and correct.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>➢ Language is more English and math like</td>
<td></td>
</tr>
<tr>
<td></td>
<td>➢ Programmer doesn’t need to know details of the computer.</td>
<td></td>
</tr>
</tbody>
</table>
# GENERATION OF LANGUAGES

<table>
<thead>
<tr>
<th>GENERATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FIRST GENERATION</strong></td>
<td>Machine language was used during this period</td>
</tr>
<tr>
<td><strong>SECOND GENERATION</strong></td>
<td>Assembly language was introduced and used</td>
</tr>
<tr>
<td><strong>THIRD GENERATION</strong></td>
<td>Earlier high level languages such as FORTRAN, BASIC and COBOL became popular</td>
</tr>
<tr>
<td><strong>FOURTH GENERATION</strong></td>
<td>These are the newest breed of programming languages. They are often described as very high level languages and referred to as 4GLs. 4GLs are easy to learn, easy-to-use languages that enable users or programmers to code applications much more quickly than they could with lower level languages. Two examples are DBASE and FOXPRO</td>
</tr>
</tbody>
</table>
SOLVING THE PROBLEM

The most important part of a programmer’s job is solving the problem first. You should first think about a method and develop an ALGORITHM to solve the problem.

An ALGORITHM is a sequence of precise instructions, which results in a solution. Algorithms may be written using special statements and rules. These statements are called PSEUDOCODES. Pseudocode is a language similar to a programming language but the rules are not as stringent. Each statement must have the correct syntax. The syntax is the arrangement of words in an instruction.

ALGORITHM DEVELOPMENT

This consists of three steps:

1. Clearly define the problem that you want to solve.
2. Design an algorithm that is precise and well thought out to solve the problem.
3. Test your algorithm on paper. Make sure your algorithm works correctly before you can write a program for it.

DEFINING THE PROBLEM

When you first analyse a problem, you should specify its objectives, this is, what the program is meant to do. The following steps are used to organise and summarise a program’s objectives:

1. Specify the output: what kind of output are you expecting from this program? Is it in a readable form or is it for input to another program?
2. Specify the input: since you know what the output is, you can specify what the input should be.
3. Specify the processing: what processing should be done on the input to get the necessary output?
DATA AND DATA TYPES

Data is facts and figures that can be categorised as:

1. Data that is always the same value and does not change, for example:
   - The number of minutes in an hour
   - The speed of light

2. Data that changes or varies, for example:
   - The price of an item
   - The age of a person

Both categories of data have a number of data types.

CONSTANTS

Data that does not change is called a CONSTANT. Some constants may be numbers while others may be a series of characters.

CHARACTER CONSTANTS

Characters that are not likely to be calculated must be enclosed in quotation marks to distinguish them from numeric data. Such data is called a string. Example:

- ‘Jones’
- ‘11th Avenue, Cave Hill’

When the computer processes a character constant, its data remain unchanged.

NUMERIC CONSTANTS

Numeric constants are numbers that can be calculated. Two types of numbers are being considered. INTEGERS, which are whole numbers, e.g. 1, 3, 7 -89 and REAL numbers which are numbers with a decimal point, e.g. 23.5, -34.76

VARIABLES

A VARIABLE is a name that represents a piece of data that can take many possible values. If the variable is used in the entire program then it is called a GLOBAL VARIABLE. If it is used in one part of the program, then it is called a LOCAL VARIABLE.
DATA INPUT AND STORAGE

Data to be processed must be entered into the computer and stored. Such data may be entered from the keyboard.

When developing the Pseudocode for solving a problem, the commands that are used to permit the input of data are the words **READ** or **INPUT**. This is followed by one or more variable names to represent the data that is being enters and stored. When more than one variable is used, place the comma between each variable.

The **SYNTAX** is:

**READ <variable name>, <variable name>**

**EXAMPLE:**

Write an instruction to input the quantity and price of an item.

**SOLUTION:**

**READ QUANTITY, PRICE**

Write an instruction to read three scores.

**READ SCORE1, SCORE2, SCORE3**

**PROMPTS**

While entering data, messages may appear on the screen notifying the user of what data is to be entered. These messages are called **PROMPTS**. **PROMPT** statements begin with the commands **PRINT**, followed by the message enclosed in quotes.

**EXAMPLE:**

**PRINT “Enter the Name”**

*A READ statement usually follows the prompt statement to facilitate the entry of data corresponding to the prompt.*

**EXAMPLE:**

**PRINT “Enter the Name”**

**READ Name**
PRINTING INFORMATION

THE OUTPUT INSTRUCTION

When developing the Pseudocodes for solving a problem, the command that is used to produce the output is the word PRINT.

PRINT can be used to output the value of a variable or to output the data that is constant.

OUTPUTTING THE VALUE OF A VARIABLE

The SYNTAX is:

PRINT <variable name>

EXAMPLE:

PRINT SUM

When a variable is printed, the content of the variable is printed and not the name of the variable.

EXAMPLE:

A=10
PRINT A

In this example, the variable A is given the value 10 as in the first instruction. The second instruction causes 10 to be printed and not the letter A.

OUTPUTTING A CONSTANT

The SYNTAX is:

PRINT <string>

EXAMPLE:

PRINT “NAME”

PROCESSING INVOLVING CALCULATIONS

CALCULATIONS USED FOR PROCESSING DATA

Calculations can be done by using the mathematical operations for addition, subtraction, multiplication, division i.e. +,-,*,/
The results of the calculation must be stored in a variable for future use, i.e. for printing or use in subsequent calculations. The syntax for a calculation instruction is:

\[ \text{<VARIABLE>} = \text{<WHAT IS BEING CALCULATED>} \]

**EXAMPLE:**

\[ \text{AmtDue} = \text{Price} \times \text{Quantity} \]

Any statement that assigns a value to a variable is called an **ASSIGNMENT STATEMENT**. The variable, which begins an assignment statement, should not be a variable that was read as input if the input is to be printed.

**PROCESSING INVOLVING SELECTING INSTRUCTIONS**

**FORMING CONDITIONS**

When a program is executed each instruction is processed in the sequence listed in the program, unless specific instructions direct it to deviate and select other instructions. An instruction containing the command IF allows deviation and selection to take place. Making a comparison based on a condition and acting on the result achieves this requirement.

A condition is made up of three parts. The first part is a variable that was carried out before.

The second part is a **RELATIONAL OPERATOR**, and the third part is a variable that was carried out before or a constant.

**The relational operators are:**

\[ =, \neq, >, <, \leq, \geq \]

A condition is evaluated when the instruction containing it is carried out. When the condition is met, the condition is said to be **TRUE**, if the condition is not met, then the condition is said to be **FALSE**.
CHOOSING THE CORRECT IF CONSTRUCT TO SOLVE THE PROBLEM

A number of IF constructs are available for use in programming. Each construct will perform differently; hence it is important to know what results you are trying to accomplish in order to choose the appropriate construct.

When ONE option is available and a selection may or may not be made, use the IF-THEN CONSTRUCT.

When TWO OR OPTIONS are available and a selection must be made, use the IF-THEN-ELSE CONSTRUCT.

When TWO OR MORE OPTIONS are available and a selection may or may not be made, use the IF-THEN-ELSE-IF CONSTRUCT.

THE IF-THEN CONSTRUCT

The syntax for the IF-THEN construct is as follows:

IF <Condition> THEN

<One or more instructions which will be carried out if the condition is TRUE>

ENDIF

EXAMPLES:

Read the time. If the time is 11:00, output “Ring the Bell”
PRINT “ENTER THE TIME”
READ TIME
IF TIME = 11 THEN
   PRINT “RING THE BELL”
ENDIF

Read a number N. If N is greater than 100, add 10 to the number. Print the number.

PRINT “ENTER A NUMBER”
READ N
IF N > 100 THEN
   RESULT = N +10
ENDIF
PRINT “NUMBER”, N

Print the sum of three numbers if the sum is greater than 50.
READ N1, N2, N3
SUM = N1 + N2 + N3
IF SUM > 50 THEN
   PRINT TOTAL
ENDIF
THE IF – THEN – ELSE CONSTRUCT

The syntax for the IF-THEN-ELSE construct is as follows:

   IF <Condition> THEN
       <One or more instructions which will be carried out if the condition is TRUE>
   ELSE
       <One or more instructions which will be carried out if the condition is FALSE>
   ENDIF

EXAMPLES:

Input the age of a person. If the age is greater than 35, output “old person” otherwise output “young person”

PRINT “ENTER THE AGE”
READ AGE
IF AGE > 35 THEN
   PRINT “OLD PERSON”
ELSE
   PRINT “YOUNG PERSON”
ENDIF

A student is given a 5% discount off the fees for a course if the fees are paid before 30 days. Read a fee and the number of days. Output the fee, discount amount and fee less the discount amount.

PRINT “ENTER THE FEE AND THE NUMBER OF DAYS”
READ FEE, DAYS
IF DAYS < 30 THEN
   DISCOUNTAMT = FEE * 5/100
ELSE
DISCOUNTAMT = 0
ENDIF
AMTDUE = FEE – DISCOUNTAMT
PRINT “FEE”, FEE
PRINT “DISCOUNT AMT”, DISCOUNTAMT
PRINT “FEE LESS DISCOUNT AMT”, AMTDUE

THE IF – THEN – ELSE -IF CONSTRUCT
The syntax for the IF-THEN –ELSE- IF construct is as follows:

IF <Condition> THEN
  <One or more instructions >
ELSE
  IF <condition> THEN
    <One or more instructions>
  ENDIF
ENDIF

EXAMPLE:
A stadium has four stands, A, B, C and D. The admission fee for stand A is $2.00, stand B is $2.50, stand C is $4.00 and stand D is $5.00. Read a stand and the number of spectators in the stand. Calculate and print the revenue for the stand.

PRINT “ENTER A STAND AND THE NUMBER OF SPECTATORS”
READ STAND, SPECTATORS
IF STAND = “A” THEN
  REVENUE = SPECTATORS * 2.00
ELSE
IF STAND = “B” THEN
    REVENUE = SPECTATORS * 2.50
ELSE
IF STAND = “C” THEN
    REVENUE = SPECTATORS * 4.00
ELSE
IF STAND = “D” THEN
    REVENUE = SPECTATORS * 5.00
ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
PRINT “STAND”, STAND
PRINT “REVENUE”, REVENUE

N.B. FOR EVERY IF STATEMENT THERE MUST BE A CORRESPONDING ENDIF

LOOPS

Loops are useful for repeating parts of a program. That is, they will repeatedly execute a section of a program until the end condition is satisfied. In order to exit from a loop, you must have a method for checking to see if you have completed the task. Once a loop terminates, control is returned to the first sentence after the block of sentences in the loop.

The basis structure of a loop is:

- Initialise the variable to some start value – this variable usually determines whether or not the loop executes or not.
- Test the variable against a condition.
- Execute the body of the loop.
- Update the value of the variable.

There are two types of loop statements:
✓ Indefinite – when you do not know in advance how many times to repeat the loop (WHILE or REPEAT loops)
✓ Definite – when you know in advance how many times to repeat the loop (FOR loop).

**WHILE LOOP**

User input ---- (then) ---- check condition ---- (then) ---- performs statements

**REPEAT LOOP**

User input ---- (then) ---- performs statements ---- (then) ---- check condition

**WHILE - DO LOOP**

The WHILE –DO loop repeatedly executes a statement or a block of statements as long as the condition is TRUE.

The condition is a WHILE loop is tested at the beginning of the loop, so it is possible for the statement not to be executed at all.

The general form of the WHILE – DO LOOP is:

<An initial value for the condition>

WHILE <condition> DO
   <Instructions which are to be repeated>
ENDWHILE

The initial value for the condition can be stored in a variable which is read or it can be a value which is assigned to the variable. The initial value is necessary so that the comparison for the condition can be made when the WHILE instruction is executed the first time.

The termination constant is known as the **DUMMY VALUE**. A dummy value is not a real value for the problem being solved. For example 999, could be the dummy value.
EXAMPLE:

NUMBER = 1
WHILE (NUMBER <= 3) DO
    NUMBER = NUMBER + 1
    PRINT “THE NUMBER IS”, NUMBER
ENDWHILE
PRINT “OUT OF LOOP”

AGE = 15
WHILE (AGE < 20) DO
    PRINT “YOU ARE NOT 20 YEARS OLD”
    AGE = AGE + 1
ENDWHILE
PRINT “YOU ARE OUT OF THE LOOP”

SELECTING THE CORRECT OPERATOR FOR THE CONDITION

1. Use = if all values except a particular value can cause the loop to be terminated.

   EXAMPLE:
   
   WHILE NO = 0 DO
   The loop is repeated once, NO = 0, otherwise it is terminated.
   
2. Use <> if only one value is to be used for terminating the loop.

   EXAMPLE:
   
   WHILE SCORE <> 999 DO

3. Use <= or < if the value to terminate the loop are greater than the dummy value or greater than or equal to the dummy value respectively.

4. Use >= or > if the values to terminate the loop are less than the dummy value or less than or equal to the dummy value respectively.
COMPARISON OF THE DIFFERENT LEVELS OF LANGUAGES

MACHINE LANGUAGE:

ADVANTAGES:
Programs execute fast. Efficient use of memory.

DISADVANTAGES:
Programming slow and tedious. Code difficult to learn, read, understand & correct. Code is machine specific i.e. can’t be used in the same form on other computer models.

ASSEMBLY LANGUAGE:

ADVANTAGES:
Programming faster & less tedious than for machine language. Code is easier to learn and understand for machine language. Execution faster than high level languages. More compact than high level languages.

DISADVANTAGES:
Programming slower & more tedious than for high level languages. Code is machine specific.

HIGH LEVEL LANGUAGES

ADVANTAGES:
Programming faster and less tedious. Code is easier to learn, read, understand & correct. Language is more English & math like. Programmer doesn’t need to know details of the computer.

DISADVANTAGES:
Programs executes slower.

NATURAL LANGUAGES
These are still in a stage of infancy. They enable users to communicate with computers in their native language (English, French, etc.)

**GENERATIONS OF LANGUAGES**

Computer languages are seen to have evolved over four generations of the computer age as listed below:

1. **FIRST GENERATIONS**: Machine languages were used during this period.
2. **SECOND GENERATION**: Assembly languages were introduced and used.
3. **THIRD GENERATION**: Earlier High level languages such as FORTRAN, BASIC and COBOL became popular.
4. **FOURTH GENERATION**: These are the newest breed of programming languages. They are often described as high level languages and referred to as 4GLs, i.e. fourth generation languages. 4GLs are easy to learn, easy to use languages that enable users or programmers to code applications much more quickly than they could with lower level languages. Tow examples are DBASE and FOXPRO.

**CREATING PROGRAMS**

The general steps to follow when writing a program:

1. Define the problem that the computer is to solve.
2. Analyze the problem.
3. Develop an algorithm to perform these steps and test it.
4. Write the computer program corresponding to the algorithm.
5. Test and debug the program.
6. Document how the program works and how to use it.

**EXAMPLE TO ILLUSTRATE HOW THIS WORKS:**

1. *Define the problem:*
   a) The computer asks the user to enter 3 numbers.
   b) The computer adds these 3 numbers.
   c) The computer prints the total.

2. *Analyze the problem:*
   Here we ensure we have a clear understanding of what is required. We then look at the various options available to achieve this and determine which is best. For example, we may determine the inputs are the 3 numbers and the output is the total.

3. a) Develop the algorithm
   An Algorithm is a sequence of steps designed to perform a particular task. Algorithms can be written in any suitable form such as in a programming language or in PSEUDOCODE. Pseudocode is a very popular form of describing algorithms because it is easy to write, understand and translate to the actual programming language being used to write the program.

   Example using pseudocode:
Set total to 0

For counter from 1 to 3

Get the next number

Add number to total

Next

Print the total

b) Test the Algorithm:

The algorithm must be checked by doing a DRYRUN, i.e. by working through the steps manually. This is useful or locating errors before the program is actually created. TRACE TABLES are tools which are very helpful during dry runs for testing the logic and correctness of algorithms. Example of trace tables, the numbers 11, 25 and 37 are used as the input data, i.e. the numbers to be added.

<table>
<thead>
<tr>
<th>STEP</th>
<th>TOTAL</th>
<th>COUNTER</th>
<th>NUMBER</th>
<th>TOTAL</th>
<th>COUNTER</th>
<th>NUMBER</th>
<th>TOTAL</th>
<th>COUNTER</th>
<th>NUMBER</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>2</td>
<td>2</td>
<td>25</td>
<td>2</td>
<td>2</td>
<td>25</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>3</td>
<td>3</td>
<td>36</td>
<td>3</td>
<td>3</td>
<td>36</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
4. Write the program.
In this step we simply translate each step in the algorithm into the program code using the chosen programming language:

```
Total = 0

For counter = 1 to 3 Do
    Read number
    Total = total + number
Next

Print total
```