CMP 121: INTRODUCTION TO BASIC PROGRAMMING (2 UNITS)

COURSE OUTLINE

Types of Programming languages, Introduction to BASIC, Constants and Variables, Control Structures, Arrays, Functions and subroutines, Data Files and Introduction to Computer Graphics.

OVERVIEW OF PROGRAMMING

Programming Can be defined as the practice of designing, coding, testing and implementing program codes intended or tailored to solve a desired task (problem). Our use of the computer obviously is limited by the nature and type of software and packages available, besides this fact, even when the appropriate package is available it may not completely meet our specific demand. Hence one can then write (develop) his or her own program (software) to satisfy his or her peculiar problems as a result, programming is indispensable especially among professionals.

Computer program

A computer program is a set of instructions that tell a computer exactly what to do. The instructions might tell the computer to add up or set of numbers or compute two Numbers and make a decision based on the result or whatever. But a computer program is simply a set of instructions. The computer follow your instructions exactly and in the process does something useful like generating an employee payroll or display a game on the screen or implementing a word process.

A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

The functions of a computer system are controlled by *computer programs; a* computer program is a clear, step-by-step, finite set of instructions. A computer program must be clear so that only one meaning can be derived from it. A computer program is written in a computer language called a *programming language.*

CATEGORIES OF PROGRAMMING LANGUAGE

There are three categories of programming languages:

- 1. Machine languages.
- 2. Assembly languages.
- 3. High-level languages.

Machine languages and assembly languages are called low-level languages. *A Machine language program* consists of a sequence of zeros and ones. Each kind of CPU has its own machine language.

- Advantages
  - Fast and efficient
  - Machine oriented
  - No translation required
- Disadvantages
  - Not portable
  - Not programmer friendly

Assembly language programs use mnemonics to represent machine instructions. Each statement in assembly language corresponds to one statement in machine language. Assembly language programs have the same advantages and disadvantages as machine language programs. Compare the following machine language and assembly language programs:

8086 Machine language program for var1 = var1 + var2	8086 Assembly program for var1 = var1 + var2

1010 0001 0000 0000 0000 0000	MOV AX , var1
0000 0011 0000 0110 0000 0000 0000 0010	ADD AX , var2
1010 0011 0000 0000 0000 0000	MOV var1 , AX

A high-level language (HLL) has two primary components

- (1) a set of built-in ***language primitives*** *and* ***grammatical rules***
- (2) a **translator**

A HLL language program consists of English-like statements that are governed by a strict *syntax*.

- Advantages
  - Portable or *machine independent*
  - Programmer-friendly
- Disadvantages
  - Not as efficient as low-level languages
  - Need to be translated
- Examples : C, C++, Java, FORTRAN, Visual Basic, and Delphi.

Types of Programming language

In order for computer to recognize the instruction you give it, those instruction need to be written in a langrage, the computer understands a programming language, there are Many computer programming languages Fortran. COBOL, basic, Pascal, C, C++, java, etc just like there are many spoken languages. They all express approximately the same concepts in different ways.

However, it is important to note that the computer understands only the machine language (strictly binary codes), this implies that all programs written needs to be converted/ translated into machine code. These conversions are done by special programs (software) called **TRANSLATORS.**

Typically we have **compilers, interpreters and assemblers.**

INTERPRETER:- This is a software (program) which translates program codes written in any programming language into machine readable codes (machine codes). The interpreter thus interprets and executes each program statement one at a time. Examples of interpreter's languages are: BASIC and some versions of LISP a programming language used mainly in the area of artificial intelligence

Compiler: just like the interpreter, a compiler is a special software which translates written program codes into machine language, but unlike the interpreter, the compiler translates all program statements (compilation), before any executions can begin. FORTRAN, PASCAL and COBOL programming languages use compilers.

SOURCE CODE: This is a term which refers to the original codes (program statements) written in a particular programming language say FORTRAN, BASIC, PASCAL, etc

OBJECT CODE: This refers to the interpreted (for interpreters) or compiled (for compilers) source codes. In other words, object code simply refers to the translated source code which is now in machine language format, i.e. series of 0s and 1s

BUG: A bug in a computer programming refers to an error in coding which could be either logical, syntactic or execution errors.

DEBUGGING: This is the process of removing the existing bugs (errors) from your program (i.e. the process of correcting your program codes to produce the desired results).

STAGES OF PROGRAM DEVELOPMENT (Programming Process)

Before a computer program is successfully written, it must have passed through the following stages.

1. **Problem Definition:**

Before any useful program could be written, the problem that prompted it must have to be defined. No one solves a problem he does not understand. The problem to be solved by computer should be well stated and understood before the solution will be worked out.

Knowing the objective is the first consideration. Is it a payroll or editing program? Knowing who the end user will be is also important. Determining the inputs and outputs is next. How will the program operate and what data is needed to make it happen. After this has been decided, feasibility is the next consideration. How many programmers will it takes, is the project within budget, does the project have a realistic outlines, finally, if the project is good, then one must take measures to ensure the project is properly documented and analyzed.

Six mini steps

- Clarify objectives and users
- Clarify desired output
- Clarify desired input
- Clarify desired processing
- Double check the feasibility of implementing the program.

2. **Development of algorithm**

An algorithm is set of well defined rules for solving a problem in a finite number of steps or simply a description of the step by step method of solving a problem. Each step in the algorithm must be made perfectly clear and void of assumption or vague ideas. It must contain a finite or limited number of operations. It should be able to provide output in order to know the result of implementing the procedures. It should have an existing point where the operation ends. **EXAMPLES OF ALGORITHM**

1. Write an algorithm to compute and print out the average of three numbers a, b, c,

Solution:-

1. Get the numbers a, b, c
2. Compute sum = a+ b + c
3. Compute average = sum divided by 3
4. Write average.
5. Stop.

2. An Algorithm to calculate the area of a room

Solution:-

3. Get length
4. Get width
5. Compute area = length multiplied by width
6. Write area
7. Stop

3. An algorithm to compute managers commissions given that commission is 10% of sales provided sales is greater than N50,000 other wise no commission is given.

Solution:-

1. Get sales
2. Check if sales > N50,000 then
3. Compute commission = 10/100 multiplied by sales
4. Else commission = 0
5. Print commission
6. Stop.

Exercises

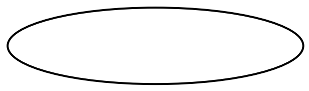
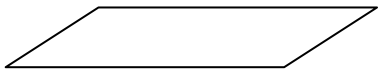
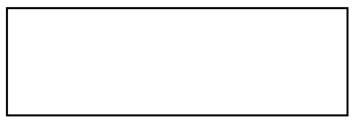
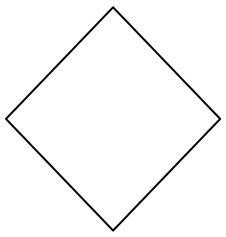
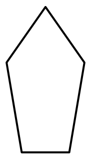
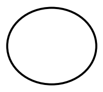
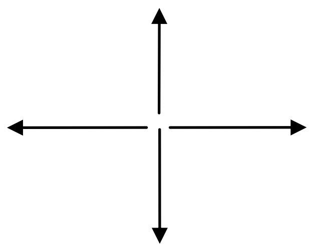
1. Write an algorithm to calculate the area of a triangle with base b and height h.
2. Write an algorithm that compare three numbers (a, b, c), compare them and print the largest.
3. Write an algorithm that will accept the scores of 30 students in a class calculate and print the average mark.

4. Write an algorithm that will accept any number and determine whether the number is odd or even.

3. **Flowcharting**

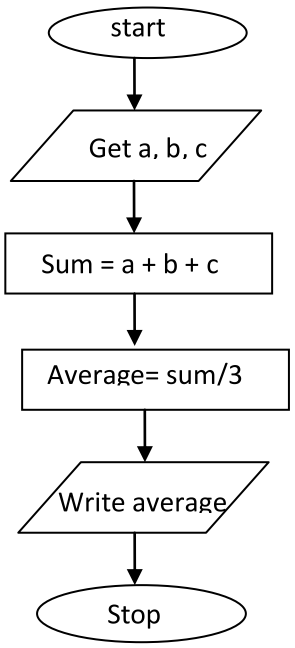
   The algorithm can be converted into either a written form or pictorial form, or both. The written form is called the pseudo code, and the pictorial form is a FLOWCHART. The flowchart is thus a pictorial representation of an algorithm. It helps the programmer in planning the sequence of operation involved in his program before actually writing the program in a programming language.

A flow chart illustrates a process graphically using a set of standard symbols, the shape of each of which indicates the operation being described. Thus, the flow chart acts as a visual aid for programming. Each symbol represents an operation and has a specified geometrical shape joined together by means of arrows. The arrows indicate the flow of logic. Each box contains outlines of the steps to be executed by the computer at each stage. Below are some of the flowcharting symbols commonly used in expressing the logic of programs.

	Start/Stop
	Input / Output
	Processing
	Decision
	Off-Page connector
	On-Page connector
	Directional arrows

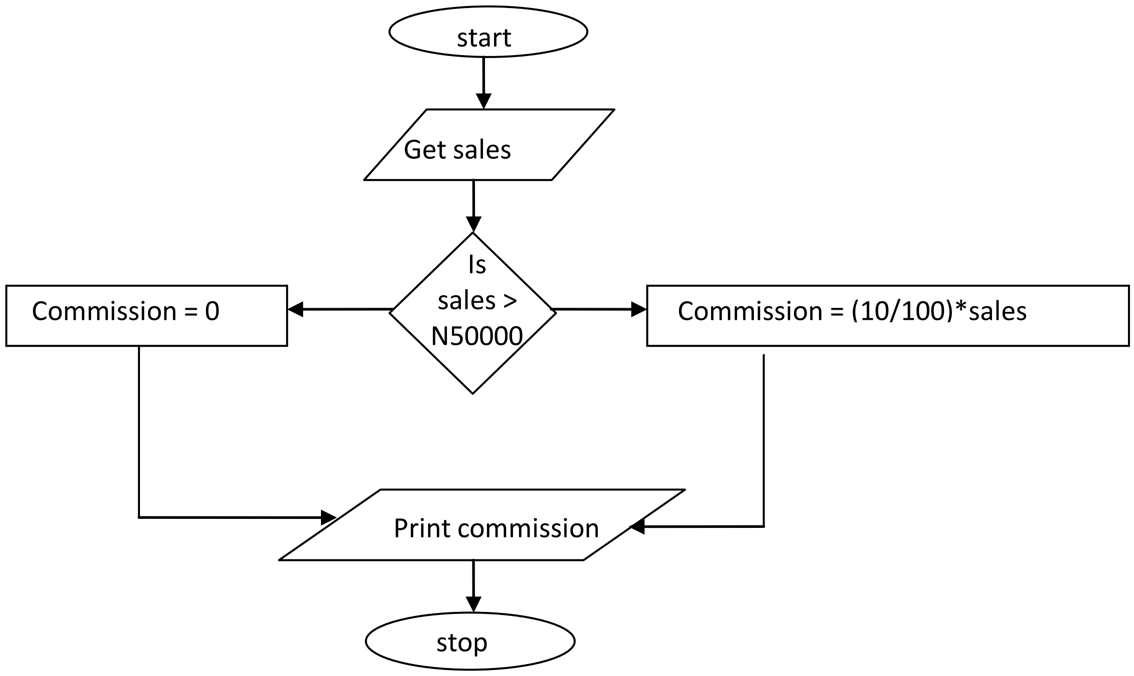
Examples of flowcharts

1. A flowchart to compute and print out the averages of three numbers a, b, c,



2. A Flowchart to compute managers commission giving that commission is 10% of sales provided sales is > N500000, otherwise, no commission is given

Solution:



4. **Code the program**

After the problem has been analyzed it must be coded or written using the desired programming language. Once the appropriate programming language has been chosen, it is imperative that the programmer follow the *syntax* rules of the programming language.

Two mini steps:

- Select the appropriate high-level programming language
- Code the program in that language following the syntax carefully

5. **Testing and Documentation**

   The essence of program testing also referred to as program validation is to determine whether any error still remain in the program. Testing is the process of running the computer program and evaluation the program results in order to determine if any errors exist. The testing is done by running the program with various sets of inputs values so as to be sure that the expected results is gotten.

   Documentation is the description of the program in the proper form for users and to enhance maintainability. It describes the working of a program and how expected problems could be solved. Documenting a program is very important because it aids the users in understanding the program better and it also aids maintenance of such program. The documentation may be internal – in the form of comments which exist within the program, or may be external in the form of a written description and structured diagrams.

INTRODUCTION TO BASIC PROGRAMMING

In computer programming, **BASIC** (an acronym for **Beginner's All-purpose Symbolic Instruction Code** is a family of high-level programming languages.

The original BASIC was design in 1964, by John George Kemeny and Thomas Eugene Kurtz at Dartmouth College to provide access for non-science students to computers. At the time, nearly all use of computers required writing custom software, which was something only scientists and mathematicians can do. The language (in one variant or another) became widespread on microcomputers in the late 1970s and home computers in the 1980s. BASIC remains popular to this day in a handful of highly modified dialects and new languages based on BASIC such as Microsoft Visual Basic.

The eight design principles of BASIC were:

1. Easy for beginners to use.
2. A general-purpose programming language.
3. Allow advanced features to be added for experts (while keeping the language simple for beginners).
4. interactive.
5. Provide clear and friendly error messages.
6. Respond quickly for small programs.
7. Not to require an understanding of computer hardware.

BASIC CHARACTER SETS

While learning a natural language, we normally start from its alphabets which are called characters. These characters are broadly classified into three categories viz Alphabetic, Numeric and special characters

i) Alphabetic characters are the 26 alphabetic characters (A, B, C, … Z)
ii) Numeric characters are the 10 numeric characters (0 1 2 3 4 5 6 7 8 9)
iii) Special characters are (+ - * / + = , . ( ) " & $? etc)

TYPES OF OPERATORS

We have three classes of operators

i) Arithmetic operators
ii) Relational operators
iii) Logical operators

Arithmetic operators: are those operators which permit arithmetic/mathematical operations/calculations to be performed on any given data (operand), they include **+, -, *, /, **, ^**

+	Addition
-	Subtraction
*	Multiplication
/	Division
** or ^	Exponentiation

Relational operators: are those operators which allow for comparison. Below are some relational operators in BASIC and their meanings

Operator	meaning
>	Greater than
<	Less than
=	Equal to
>=	Greater than or equal to
<=	Less than or equal to
< >	Not equal to

Logical operators

These are operators which allow for a selection based on one or more conditions being true or false. These operators include OR, AND and NOT

Example: IF A = 1 OR B = 2 THEN PRINT "TRUE"

A < B AND A< C THEN PRINT "LAST = A"

DATA TYPES

BASIC programming language uses two types of data:-

1. **Numeric data types** eg 10, 20, 330.50, 2.5

2. **Alphanumeric/String** data eg “gina” “ P. o. Box 2041”

All data is represented in the computer in binary form. The type of data determines the way in which it is represented and the operations which may be performed on it. Data is usually referred to as variables.

In programming, a variable is a value (data) that can change, depending on conditions or on information passed to the program. Typically, a program consists of instructions that tell the computer what to do and data that the program uses when it is running. The data usually consists of ***constants*** or **fixed values** that never change and variable values (which are usually initialized to "0" or some default value because the actual values will be supplied by a program's user). Usually, both constants and variables are defined as certain data types.

Reserved words (occasionally called Keywords) are special words reserved by a programming language or by a program. You are not allowed to use reserved words as variable names. For example, in BASIC the words Print, Remark, Input, Locate, Data, etc are reserved because they have special meaning.

A **variable** is usually represented with a unique name which is called **variable name**. Program applies such a name to memory and uses it to refer to it. A variable name usually consists of alphabetic characters and decimal digits, beginning with an alphabetic character. It is usually good to give meaningful names to your variables. Examples: VOL, TEMP, A2, COLUMN, IBM370 etc. Names are given to memory locations for easy referencing. A variable name in BASIC consists of 1 to 40 characters. The string variable name always have dollar ($) sign attached to it as the last character.eg name$. It cannot be used in a formula during computation. The numeric variable can be used in computation, it does not have dollar sign ($) attached to it. E.g. Length, amount, price. It is advisable to give meaningful name to your variable. **Reserved words** should not be used in forming variable names.

Rules for forming variable names in Summary.
1. All variable names must start with a letter of the alphabet from A to Z.
2. No special symbols or basic key words must be used as variable name.
3. If more than one variable is used in a computer program, each variable name must be different.
4. Keep names short and meaningful as to their use.

5. Variable names should help you remember what they hold.
Good examples: age, salary, name, weight, etc.
Bad examples: a, b, c, x, y, z.
6. The variable type or usage must be declared or specified by:
a) use a name (with no special symbol) for a NUMERIC location.
b) use a name with a $ sign at the end for an ALPHANUMERIC location.

RULES FOR WRITING BASIC PROGRAMS

1- It must start with a line number (number statement)
2- Two statements can be combined in a line, but should be separated by comma
3- BASIC statements should always follow the number statement
4- Succeeding lines must be greater than the preceding lines
5- Numbers must be whole numbers
6- No repetition of number statement
7- There should be at least a space between the number statement and BASIC statement
8- A full stop is no allowed at the end of any statement the keyboard when the program
9- The program must always stop with an END statement.

BASIC STATEMENTS

1- The **REM** statement

   REM: means remark to basic interpreter. It is used to indicate comment line. In a program. It can be replaced with a quote mark. Example:

   10 Rem to sum three numbers

   10 ‘To Sum Three Numbers

2- **INPUT** Statement

It is used to ask for input (data) through the keyboard when the program is running. It is used for interactive programming. INPUT Statement is followed by variable name, more than one variable name can declared with an input statement but should be separated by comma. When running the program the data should also be typed separated by comma as declared in the INPUT statement. E.g.

```
INPUT AMT, NAME$, discount
```

3 **THE READ AND DATA** Statement

This is another Statement used to ask for input (data), but this time the data is not expected from the keyboard but within the program. When the BASIC interpreter encounters a READ statement, it looks for the DATA statement which contains the actual data for the variables declared in the READ statement. The data statement must be used anytime the READ statement is used. The DATA statement must match with the READ statement variables declared.

```
e.g. READ NAME$, AMT

       – –

      – –

     DATA ”Bag “, 30000
```

4 THE **PRINT** Statement

This is an output statement, it is used to tell the computer to return to us or show us the results from computation on the screen. Example: 10 PRINT A.

Where A is a variable name, more than one item may be listed in the print statements but they must be separated by commas or semicolons.

Example of BASIC programs using the above statements

1. A program to calculate the area of a room
   Solution

```
10 REM A PROGRAM TO CALCULATE THE AREA OF A ROOM
20 INPUT "ENTER THE LENGTH"; LENGTH
30 INPUT "ENTER THE WIDTH"; WIDTH
40 LET AREA = LENGTH * WIDTH
50 PRINT " THE AREA OF THE ROOM IS"; AREA
60 END
```

2. To rewrite the program using the READ/DATA Statement

   Solution

```
10 REM A PROGRAM TO CALCULATE THE AREA OF A ROOM
20 READ LENGTH, WIDTH
30 LET AREA = LENGTH * WIDTH
40 DATA 100, 50
60 PRINT " THE AREA OF THE ROOM IS"; AREA; "M2"
70 END
```

5. **LET** STATEMENT: The LET statement is used for assignments. The LET statement operates from right to left. That is to say, the value on the right of the equal sign is assigned to the variable on the left of the equal sign. The sign means an assignment and not equality. Example

let PI = 3.142,   let sum = A + B + C

1. Program that accept three numbers (A, B, C), compare them and print the largest.

Solution

```
10 REM A PROGRAM THAT ACCEPTS THREE NUMBERS COMPARE THEM AND PRINT THE LARGEST

20 INPUT "ENTER THE NUMBERS "; A, B, C

30 IF A > B THEN BIGGER = A

40 ELSE BIGGER = B
```

```
50 IF BIGGER > C THEN BIGGEST = BIGGER

60 ELSE BIGGEST = C

70 PRINT BIGGEST

80 END
```

6. **GOTO** STATEMENT: is used to transfer execution to a specified line number. Example GOTO 20- this tells the computer to continue execution from line 20.
7. **IF – THEN – ELSE** STATEMENT

This statement is used to test whether a transfer of control or execution will take place or not. It is a powerful statement that gives the computer the ability to make decisions and take and alternative action based on the out come. It is always used in conjunction with relational operators to set the condition that will be tested.

1. A program that will read in the marks scored by a student in six different courses and calculate the average score.

Solution

```
10 I = 0.0

20 SUM = 0.0

30 INPUT SCORE

40 I = I +1

50 SUM = SUM + SCORE

60 IF I <= 5 THEN GOTO 3O

70 AVERAGE = SUM/6

80 PRINT AVERAGE

90 END
```

NOTE: the ELSE statement is not mentioned in the above program, but line 70 is by default the implied ELSE statement because it is only executed when line 60 is no longer true.

8. **DIM** STATEMENT

   This statement is used for declaration of array or matrix, it must be the first statement to be coded before any statement that reference that array or matrix. Array can be single dimensional or double dimensional.

   Example:

   DIM A(10)- This is a single dimensional array with 10 elements.

   DIM B(5,4) – This is a double dimensional array or matrix named B with 5 rows and 4 columns given a total of 20 (i.e. 5 * 4) elements in the array.

   1. A program that will read in 20 elements of an array called A and calculate their average.

      Solution

      ```
      10 DIM A(20)
      20 FOR I = 1TO 20
      30 INPUT A( I )
      40 SUM = SUM + A(I)
      50 NEXT I
      60 AVERAGE = SUM/20
      70 PRINT AVERAGE
      80 END
      ```

   2. A program to read and print the elements of a 3 x 4 matrix.

      Solution

      ```
      10 DIM A (3,4)
      20 I = I + 1
      30 J = J + 1
      40 INPUT A(I,J)
      50 PRINT A(I,J)
      60 IF J <= 4 THEN GOTO 30
      70 J = 0
      80 IF I <=2 THEN GOTO 20
      ```

```
90 END
```

9. **CLS** STATEMENT

   This is a statement used to clear or tidy up the screen. Example:

   10 CLS –This clears the previous content of the screen or display.

EXPRESSIONS

EXPRESSION: an expression is a rule which when evaluated will give a single value, in its simplest form it consist of two operands with an operator between them. In basic there are numeric expression, logical expression, and string expression.

NUMERIC EXPRESSION:- to understand numeric expression let us look at arithmetic operators, arithmetic operators are used to indicate arithmetic operation such as addition, subtraction, division, multiplication and exponentiation.

ARITHMETIC OPERATORS

Addition:	+	(Plus sign)
Subtraction:	-	(Minus sign)
Division:	/	(Slash)
Exponentiation:	^	(Caret , or Upward-pointing arrow)

Therefore, these arithmetic operators are used to connect numeric constants and numeric variables to form ***Numeric expression***.

Examples of numeric expressions are:

```
A + B + C - D          A + B               ( 5 * P – 2 * Q ) / ( X + Y )
COUNT + 1              3.142 * R ^ 2       4 * PI * RADIUS ^ 3 / 3
```

“Many versions of BASIC include two additional arithmetic operators: integer division (\) and integer remainder (MOD). In integer division, each of the two given numbers is first rounded to an integer; the division is then carried out on the rounded values and the resulting quotient is truncated to an integer. The integer remainder operation provides the remainder resulting from an integer division”.

Examples

13 / 5 = 2.6	13 \ 5 = 2	13 MOD 5 = 3
8.6 / 2.7 = 3.185185	8.6 \ 2.7 = 3	8.6 MOD 2.7 =0
8.3 / 2.7 = 3.074074	8.3 \ 2.7 =2	8.3 MOD 2.7 = 2

HIERACHY OF OPERATIONS

There are some situations when several operators appear in an expression, to solve these problems we therefore apply hierarchy of operations rule. The hierarchy of operations is

1. Exponentiation: all exponentiation operations are performed first.
2. Multiplication and division: they have the same hierarchy, it means any one that comes first will be evaluated before the other one, but they are performed after exponentiation.
3. Integer division: in those version of BASIC that include this operation, integer division operations are carried out after all multiplication and (ordinary) division operations.
4. Integer remainder: in those version of BASIC that include this operation, integer remainder operations are carried out after all integer divisions operations.
5. Addition and subtraction:-they have the same hierarchy but all the above operations must be performed first.

USE OF PARENTHESES

It is possible to change the normal hierarchy of operations in a numeric expression. This is easily done by inserting pairs of parentheses at the proper place within the expression.

The operation within the inner most pair parentheses will be performed first followed by the operations within the second innermost pair of parentheses and so on.

Note: you have to make sure that the left and right parentheses balance.

Example

Evaluate the algebraic term $[2 (a + b)^{2} + (3c)^{2}]^{m (n+1)}$ into BAISC equivalent.

Solution

(2 * (a + b)^ 2 + (3 * c)^ 2) ^(m / (n+1))

Or $\Bigg\}$ both expressions are correct

((2 * ((a + b) ^ 2)) + ((3 * c) ^2)) ^ (m / n + 1))

SPECIAL RULES OF NUMERIC EXPRESSION

Sometimes special problems can arise if an expression is not correctly written. It is possible to avoid such problems if the following rules are applied.

1. Preceding a variable by a minus sign is equivalent to multiplication by -1.
   Example –a ^ 2 is equivalent to – (a ^ 2) or – 1* (a ^ 2).
2. Except for the first condition described, operations cannot be implied.
   Example the algebraic expression 3 (k1 + 3k2) will be written in BASIC as 3 * (k1 + 3 * k2) note that the multiplication must be shown explicitly.
3. In an expression involving exponentiation, a negative quantity can be raised to a power only if the exponent is an integer. (do not confuse the exponent in a numeric expression with the exponent that is part of a floating point constant).
   Example:
   - Consider the expression (P1 + P2) ^ 3 this is correct, because no exponent within parentheses.

   - (B ^ 2 – 4 * A * C) ^ 5 it will be valid only if the expression represents a positive quantity within the parentheses because of the exponent B^ 2.

STRING EXPRESSION

“Numeric operations cannot be performed on string constants or variables. However, most versions of BASIC allow strings and string variables to be concatenated (i.e combined, behind the other). The 1987 ANSI standard defines the ampersand (&) as a string concatenation operator though some versions of BASIC use the plus sign for this purpose.”

```
A$ = “ ONE ”
B$ = “ BILLION”
```

Then the string expression

```
A$ & “    ” & B$ &“ NAIRA ”
```

Will cause the three individual strings to be concatenated, resulting in the single string.

ONE BILLION NAIRA

In Microsoft BASIC , the string expression would be written as

```
A& + “   ” + B & + “ NAIRA”
```

<u>CONTROL STRUCTURES</u>

In simple BASIC problem we must have seen that statements were executed in the order in which they appeared within a program. Each statement is executed once and only once. Program of this kind do not include any logical control structures. i.e determine if various condition are true or false, also they do not require that certain groups of statements be executed on selective basis or executed repeatedly. Therefore conditional execution, looping (conditional and unconditional looping), selection will be discussed in control structures.

RELATIONAL OPERATORS AND LOGICAL EXPRESSIONS

To carry out conditional branching operations in basic, the following relational operators are used.

RELATIONAL OPERATORS

Equal:	=	Not equal:	<>	Less than :	<
Less than or equal:	<=	Greater than:	>	Greater than or equal:	>=

LOGICAL EXPRESSIONS

LOGICAL EXPRESSION: is an expression which use logical operators to compare numeric quantities (i.e constant, numeric variables or numeric expressions) or string that returned either TRUE or FALSE.

Note that the operands within a logical expression must be of the same type i.e both must be numeric or both must be strings.

Examples

$$\left.\begin{array}{l} \text{A = 10} \\ \text{Score > 56} \\ \text{X < 14} \\ \text{Balance <> 1000} \end{array}\right\} \text{Logical expression involving Numeric quantities}$$

Examples

$$\left.\begin{array}{l} \text{Name\$ = “ AHMED ”} \\ \text{Char\$ <> “ M ”} \\ \text{Target\$ < City\$} \end{array}\right\} \text{Logical expression involving strings}$$

Note that the string expressions involving operators <, <=, >, >= refer to alphabetical ordering, that is these operators are interpreted as” comes before “ or “ comes after ” rather than “ Less than or greater than “ the actual alphabetic ordering is determined by the system used to encode the characters (**ASCHI CHARACTER SET)**.

Upper-case characters precede lower case characters. Blank space precedes nonblank characters, string comparism carried out on a character by character basis from left to right.

Examples

Car precedes far, LOW preceded low, joy precede joys

LOGICAL OPERATORS

The logical operators in BASIC are AND, OR and NOT. The first two operators (AND and OR) are used to combine logical expressions, thus forming more complex logical expression (from true to false).

Examples of logical expression that make use of logical operators.

A = 25 AND NAME$ = “ Ahmad ”

Score > 50 OR Score AND Name$ < = “ Ahmad ”

Score < 50 OR Score <> 40

Not (Name$ = “ Ahmad ”) AND (ACCOUNT$ = “ CURRENT”)

NOTE : there are some versions of BASIC that include additional operators **XOR, EQV, IMP.** **XOR** will result in a condition that is true only if one expression is true and the other is false; **EQV** will result in a condition that is true if both expressions have the same logical value (either both true or both false); **IMP** will result in a true condition unless the first expression is true and the second is false.

<u>HIERAC HY OF ARITHMETIC, RELATIONAL AND LOGICAL OPERATORS</u>

Operation	Operators
1. Exponent	^
2. Negation (i.e., preceding a numeric quantity with a minus sign)	**-**
3. Multiplication and division	* /
4. Integer division	\
5. Integer remainder	MOD
6. Additions and subtraction	+ -
7. Relationals	= < > < < = > >=
8. Logical NOT	NOT
9. Logical AND	AND
10. Logical OR	OR
11. Logical XOR	XOR
12. Logical EQV	EQV

13. Logical IMP IMP

NOTE that within a given hierarchical group, the operations are carried out from left to right. The natural hierarchy can be altered by using parentheses as earlier explained in arithmetic expressions.

Example

Consider the logical expression

Balance > 1000 OR order = 1 AND Account$ = " Saving "

This expression is equivalent to

Balance > 1000 OR (order = 1 AND Account$ = " Saving ")

On the other hand, the logical expression below has a different interpretation.

(Balance > 1000 OR order = 1) AND Account$ = " Saving "

The explanation on the above example will be provided in the class.

<u>CONDITIONAL EXECUTION: THE IF – THEN STAEMENT</u>

IF –THEN STATEMENT is used to execute a single statement on a conditional basis.

Format/ syntax

IF logical expression (condition) **THEN**

Executable statement

Note that executable statement will be executed only if the logical expression (condition) is true otherwise, the statement following IF – THEN will be executed next.

<u>Example 1</u>

If Y >1 then

Print Y

Note that print statement will be executed only if the logical expression Y > 1 is true.

<u>Example 2</u>

IF score < = 39 then

Let remark = " Fail "

Print “ Remark ”; Remark

CONDITIONAL EXECUTION: IF – THEN – ELSE BLOCKS

An **IF – THEN – ELSE** block permits one of two different groups of executable statement to be executed on the outcome of logical test. Thus, it permits a much broader form of conditional execution than is available with a single **IF – THEN** statement.

The format/syntax:

IF Logical expression **THEN**

………………………

Executable statements

………………………….

Else

………………………….

Executable statements

………………………….

END IF

If the logical expression is true, then the first group of executable statements will be executed. Otherwise the second group of executable statements will always be executed.

Example 1

```
IF X > Y THEN

PRINT Y,” IS GREATER ”

ELSE

PRINT X, “ IS GREATER ”

END IF
```

The complete program:

program to find the larger of two numbers.

<u>**Solution**</u>

```
10 CLS

20 REM program to find the larger of two numbers

30 INPUT " Enter X and Y "; X,Y

40 IF X < Y THEN

50 PRINT X, " IS GREATER "

60 ELSE

70 PRINT Y, " IS GREATER "

80 END IF

90 END
```

<u>Example 2</u>

Write a program to calculate either the area of a circle or the area of a rectangle, base on the string assign to the variable form$.

<u>Solution</u>

```
CLS

REM program to calculate either area of a circle or rectangle

INPUT " The form "; Form$

IF ( form$ = " Circle") THEN

INPUT " Radius"; r

Let Area = 3.142 * r^2

Else

INPUT " Length and width "; L,W

Let Area = L* W

END IF

PRINT " Area = "; Area

END
```

Note that **IF – THEN** block allow a single group of statements to be executed conditionally.

<u>Example</u>

```
IF Salary >= 30000 THEN
LET Tax = 0.3 * Gpay
LET NetP = Gpay – Tax
END IF
PRINT "Gross pay =";Gpay
PRINT " TAXES"; Tax
PRINT " NET PAY"; NetP
END
```

<u>NESTED IF</u>

A more general form of the **IF- THEN –ELSE** block can be written as

IF Logical expression 1 **THEN**

……………………………

Executable statements

…………………………..

ELSEIF Logical expression 2 **THEN**

…………………………..

Executable statements

…………………………..

ELSEIF Logical expression n **THEN**

……………………………..

ELSE

…………………………….

Executable statements

…………………………

ENDIF

<u>Example 1</u>

Write a program to find the roots of the quadratic equation if b^2 - 4ac = 0,b^2 – 4ac < 0, b^2 – 4ac > 0.

Solution

```
IF b^2 – 4ac = 0 THEN
X= -b/2a
IF (b^2 – 4ac) < 0 THEN
LET  Real = -b/2a
LET imag = (4ac-b^2 ) /2a
Let X1 = Real + imag
Let X2 = Real  - imag
IF ( b^2 – 4ac ) > 0 THEN
LET ROOT = (b^2 – 4ac)$^{1/2}$
Let X1 = ( -b – Root) / 2a
Let X2 = (-b - Root) / 2a
```

The complete program

```
CLS
REM Program to calculate the root of a quadratic equation
INPUT " The value of A"; A
INPUT " The value of B"; B
INPUT " The value of c"; C
LET D = (b^2 - 4 *a *c)
IF d > 0 THEN
```

```
LET X1 = (-b + SQR(d)) / (2*a)

LET X2 = (-b - SQR(d)) / (2*a)

Print" Real root: X1= "; X1; "X2=";X2

ELSEIF

D =0 THEN

Let  X = -b / (2 * a)

Print " Repeated Root: X= "; X

ELSE

LET REAL = -b / (2 * a)

Let Imag = SQR ((-d ) /2*a )

Print " Complex Roots: X1 = "; Real; " + "; Imag ; Imag; " I "

Print "X2 = "; Real; " – " Imag; " I"

END IF
```

END

REPETITION (LOOPING)

In writing computer program it is often necessary to repeat part of a program number of times.

LOOPING:- this is the process of executing a part or segment of a program several times.

UNCONDITIONAL LOOPING:FOR – NEXT STRUCTURES

The **FOR – NEXT** Structure is a block of statements that is used to (carryout unconditional looping) execute a sequence of statement some predetermined number of times. In other word, this is a deterministic loop that executes a segment of the program for a specified number of times.

Format/syntax

```
For Variable = Initial value to final value

Statement 1 (Executable statements)

………………………………………
```

Next Variable

The initial value and the final value must be integer.

Variable is an identified whose value begins with initial value, increases by 1 each time the loop is executed, until it reaches final value. Next variable means the next value for the counter. If the next value greater than the loop it terminates.

EXAMPLE 1

Write a program to print your name twenty (20) times.

Solution

```
CLS

REM program to print your name 20 times

INPUT " Enter your name "; name$

For I = 1 to 20

PRINT " your name "; Name$

Next I

END
```

EXAMPLE 2

Write a program to calculate the average of a list of numbers.

Solution

```
CLS

REM Program to calculate the average of a list of n numbers

INPUT " Enter how many numbers "; n

LET Sum = 0

For k = 1 to n

PRINT "K = "; K

INPUT " X = "; X

LET Average = Sum / n
```

Print

Print “ Average = ”; Average

END

A more general form of the **FOR – NEXT STRUCTURE** can be written as:

FOR Variable = Initial value to final value **STEP X**

Executable statements

NEXT variable

Here X determines the amount by which initial value changes from one pass to the next. The quantity need not be restricted to an integer, and it can be either positive or negative. If X is negative, then initial value must be greater than final value (because the value assigned to)

The result will be

```
2.5          2        1.5      1          0.5          0          -0.5         -1
```

<u>CONDITIONAL LOOPING: DO – LOOP STRUCTURES</u>

A DO – LOOP Structure always begins with a DO statement and end s with a loop statement.

There are four (4) different ways to write a DO – loop structure.

1. DO WHILE …………………. LOOP
2. DO UNTIL…………………..LOOP
3. DO ……………………………LOOP WHILE
4. DO …………………………….LOOP UNTIL

1. **DO WHILE…………...LOOP**:-this is an underteministic loop that repeats the execution of a vlock of statements in a program as long as the condition (Logical expression) is true.

   General format/format

   DO WHILE Logical expression

   ………………………………..

   Executable statements

………………………………..

LOOP

<u>Example</u>

<u>solution</u>

```
LET Count = 1
Do while Count = 20
PRINT COUNT
LET Count = Count + 1
Loop
```

2. **DO UNTIL** ……………….. **LOOP:-** this is an undeteministic loop that repeats the execution of a block of statement so far the condition is false.
   General form/ format
   DO UNTIL Logical expression
   ………………………………
   Executable statements
   ……………………………..
   Loop
   <u>Example</u>
   Let us consider the example given in DO – WHILE …………….loop
   ```
   Let Count = 1
   DO Until Count >20
   PRINT Count
   Let Count = Count +1
   Loop
   ```
3. **DO ……………LOOP WHILE** :-this is a structure that continues to loop as long as the logical expression is true.
   General form/format
   DO
   ……………………….

Executable statements

……………………………..

LOOP WHILE LOGICAL EXPRESSION

<u>Example</u>

```
Let Count = 1

Do

Print Count

LET Count = Count +1

LOOP WHILE Count = 20
```

4. **DO LOOP UNTIL**:- this is the structure that loop as long as the logical expression is not true.

   General form/format

```
DO

        …………………………….

    Executable statements

        …………………………

LOOP UNTIL logical expression
```

Example

```
Let count = 1

Do

Print Count

Let count = count +1

LOOP UNTIL Count >30
```

Note: Control can be transferred out of a **DO LOOP** block using the **EXIT DO** statement. This statement is analogues to EXIT FOR, which is used with **FOR NEXT** blocks.

EXAMPLE

Program to calculate the average of a list of positive number using DO……..LOOP

Solution

```
REM program to calculate the average of a list of positivenumbers
PRINT "TO STOP "enter zero"
```

```
LET sum = 0
Let I = 1
PRINT "= "; I; "  X=";
INPUT "X"; X
DO UNTIL X = 0
LET sum = sum+x
LET I= I+1
IF sum >= 120 THEN EXIT DO
Print " I= "; I; " X = " ;
INPUT X
LOOP
LET average = sum / (I - 1)
PRINT
PRINT
PRINT" Average ="; Average
END
```

<u>NESTED CONTROL STRUCTURES</u>

Control structures can be nested (i.e embedded) one within another. The inner and outer control need not be the same. For example, an IF…..THEN block can be nested within **FOR – NEXT** structure or a **DO LOOP** structure, vice-versa. Example will be given in the class.

<u>SELECT CASE STRUCTURES</u>

This is the structure which allows one of several different groups of statement to be executed, depending on the value of an expression.

SELECT CASE STRUCTURE format

```
SELECT CASE expression
CASE value 1
Executable statements
CASE value 2
Executable statements
……………………….
CASE ELSE
Executable statements
END SELECT
```

<u>Example 1</u>

Write a program that will accept two numbers and display the value of one number to the selected power.

<u>Solution</u>

```
REM program to raise Y to a selected power
INPUT  "Y = "; Y
INPUT " X(1, 2 OR 3) = "; X
SELECT CASE X
CASE 1
       PRINT " Y  = " ; Y
CASE 2
       PRINT " Y squared = " ; Y^X
CASE 3
       PRINT " Y Cubed = " ; Y^X
CASE ELSE
    PRINT " ERROR –Please try again"
END SELECT
END
```

The expression in the **SELECT CASE** statement can be a string rather than a numeric expression. In this case, the values in the subsequent **CASE** statement must also be strings. The original string expression will then be compared with the string values in the subsequent CASE statement s until a match is found.

Example

Write a Program to choose and display string selection of different color.

```
REM  program to choose and string selection of different colour.
INPUT " Enter Colour  "; cl$
SELECT CASE COLOR$
CASE "RED"
     PRINT "Colour is RED";
 CASE "GREEN"
    PRINT "Colour is GREEN";
 CASE "WHITE"
   PRINT "Colour IS WHITE";
 CASE ELSE
PRINT " ERROR – please try again"
END SELECT
END
```

MATHEMATICAL functions

QBasic provides several functions to do mathematical calculations. A few of them are discussed here.

SQR

Use SQR to find the **"square root"** of a number.

```
PRINT SQR(1)
PRINT SQR(4)
PRINT SQR(9)
PRINT SQR(16)
PRINT SQR(25)
```

Output:

```
1
2
3
4
5
```

ABS

ABS returns the **absolute value** of a number. In other words, ABS converts a negative number to a positive number (if you pass a *positive* number, ABS does nothing).

```
PRINT ABS(12)
PRINT ABS(-12)
```

Output:

```
12
12
```

COS, SIN, TAN, and ATN

The following trigonometric functions can be done in QBasic:

COS (Cosine)
SIN (Sine)
TAN (Tangent)
ATN (Arctangent, inverse of TAN)

Example:

```
CONST PI = 3.141593

PRINT COS(PI / 4)
PRINT SIN(PI / 3)
```

Output:

```
.7071067
.8660254
```

SUBROUTINES AND FUNCTIONS

A subroutine (also called a "module") is a **"mini-program"** inside program. In other words, it is a collection of commands--and can be executed anywhere within the program. Subroutines and functions are ways to break up code into a reusable form. They allow the programmer to do a large set of operations just by calling the appropriate procedure or function. For example, to PRINT a lot of lines, such as instructions. One way to do this is to just enter all your PRINT commands directly into where you need them. At some point, you may want to move the PRINT commands, or you may use the exact set of PRINTs elsewhere in the program. To move or change all of the PRINT commands would be quite a hassle. A simpler way would be to create a procedure or subroutine and enter all of the PRINT commands there; then, when you need to execute the commands, you may simply call your procedure and it will execute each line in its content.

Differences between Subroutine and Function

A Subroutine does something and does not return anything for the programmer. For example, a procedure might be used to set the screen mode.

A function does something and RETURNS a value. For example, if you need to find the average of two values, you might write a function that takes in two numbers and returns the average.

Declaring a subroutine

A superior method of declaring a subroutine is using the SUB statement block. Under the QBasic IDE, doing so moves the SUB block to it's own section in the window to prevent accidental deletion of the module, and allows the easier organization of the program code.

Calling a Subroutine is as simple as writing the name of the function (passing any required parameters.) If you want, you can use the CALL statement to indicate to other programmers that it is a subroutine.

```
SUB NAME (params)
  ' Shared variable declarations
  ' ...
END SUB
```

Parameters passed into subroutines are passed by reference - if they are changed, their value is also changed in the calling function as well.

If you need access to a global scoped variable, they need to be declared using the SHARED keyword. To so do, Put the SHARED statement at the beginning of the subroutine, and list the variables you need to have access to.

The following example does **not** use subroutines:

```
        PRINT "Enter some text:";
        INPUT text$
        PRINT "The text you entered was: "; text$

        PRINT "Enter some text:";
```

```
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$

PRINT "Enter some text:";
INPUT text$
PRINT "The text you entered was: "; text$
```

By using a subroutine, the above program can be simplified like this:

```
CALL GetText
CALL GetText
CALL GetText
CALL GetText
CALL GetText
CALL GetText
CALL GetText


SUB GetText

    PRINT "Enter some text:";
    INPUT text$
    PRINT "The text you entered was: "; text$

END SUB
```

The following is even more concise:

```
FOR x = 1 TO 7
    CALL GetText
```

```
NEXT

SUB GetText
        PRINT "Enter some text:";
        INPUT text$
        PRINT "The text you entered was: "; text$
END SUB
```

Parameters
Parameters are numbers and strings that you pass to a subroutine, much like a QBasic command.

```
' This passes 16 as a parameter:

CALL OutputNumber(16)


' Notice the parentheses around the parameter "num."
' Any variables placed inside the parentheses are set as
' the subroutine's parameters.

SUB OutputNumber (num)

        PRINT num

END SUB
```

Output:

```
16
```

Declaring a function

A function is a form of subroutine that returns a value. Everything that applies in defining a subroutine also applies to a function. Within the function, the return value is created by using the function name as a variable - the return value is then passed to the calling expression.

```
FUNCTION NAME (params)
  ' Shared variable declarations
  NAME = result
  ' ...
END FUNCTION
```

Functions are declared in the same way as variables - it returns the variable type it's defined to return, in the same way variables are defined to contain their specified type. By default, it is a number, but appending a dollar sign indicates that it is returning a string.

Functions can only be called within an expression; unlike subroutines, they are not a standalone statement.

A function is the same as a subroutine, except it **returns a value**. Also, you must leave out the CALL command.

To return a value, set a variable with the **same name** as the function.

```
PRINT Add(10, 7)
FUNCTION Add (num1, num2)

        Add = num1 + num2

END FUNCTION
```

Output:

```
17
```

Since a function can return a value, the name of the function can end with special characters

```
' Notice the dollar sign ($) after "Add."  It means
' the function returns a string.

PRINT Add$("Hello", "World")

FUNCTION Add$ (str1$, str2$)

        Add$ = str1$ + str2$

END FUNCTION
```

Output:

```
HelloWorld
```

```
'define a function for returning a square root

  function squareRoot(value)

    squareRoot = value ^ 0.5

  end function
```

ARRAYS

An array is a list of variables of the same type. Arrays are useful for organizing multiple variables. To create an array, use the DIM (dimension) command.

The following example does *not* use arrays:

```
a = 2
b = 4
c = 6
d = 8
```

```
e = 10

PRINT a, b, c, d, e
```
Output:
```
2       4       6       8       10
```

This uses an array called vars, which contains 5 variables:
```
DIM vars(5)

' Each of these are separate variables:
vars(1) = 2
vars(2) = 4
vars(3) = 6
vars(4) = 8
vars(5) = 10

PRINT vars(1), vars(2), vars(3), vars(4), vars(5)
```
Output:
```
2       4       6       8       10
```

The above program can also be written like this:
```
DIM vars(5)

FOR x = 1 to 5
        vars(x) = x * 2
NEXT

FOR x = 1 to 5
        PRINT vars(x),
NEXT
```
Output:
```
2       4       6       8       10
```

Strings

You can also create an array of **string** variables.
```
DIM vars$(5)

vars$(1) = "Two"
vars$(2) = "Four"
vars$(3) = "Six"
vars$(4) = "Eight"
vars$(5) = "Ten"

PRINT vars$(1), vars$(2), vars$(3), vars$(4), vars$(5)
```
Output:

Two Four Six Eight Ten

SUBCRIPTS

A subscripted variable in QBASIC looks like this: A(1) and is pronounced "A Sub One". The subscript is placed in parentheses after the variable being subscripted, the subscript itself can be a variable, and can be as large as necessary. For a grogram to calculate average of 10 numbers ranging from A(1) to A(10). The values are all stored in the variable A, and the subscript could be generated with a FOR...NEXT loop.

A variable that has its data stored in subscript form is called an array.

```
FOR I = 1 TO 10

   READ A(I)

NEXT I

FOR I = 1 TO 10

   LET X = X + A(I)

NEXT I

LET X = X / 10
PRINT X
```

Reading and writing to Data files

To save data to a file:

1. Call the OPEN command, specifying the file name, file mode (OUTPUT), and file number.

2. Use PRINT, followed by the file number and the data you want to write.

3. Close the file using the CLOSE command.

The following opens a file, using mode OUTPUT and number 1, and then saves the text Hello World! to the file:

```
OPEN "testfile.dat" FOR OUTPUT AS #1
PRINT #1, "Hello World!"
CLOSE #1
```

To open a file for "reading," call OPEN and pass INPUT as the file mode. Then you can read the data by using the INPUT command.

```
OPEN "testfile.dat" FOR INPUT AS #1
INPUT #1, text$
CLOSE #1
PRINT text$
```

Output:

```
Hello World!
```

INTRODUCTION TO BASIC GRAPHICS

Practically all personal computers (PCs) allow information to be displayed as well as textually. Such displays permit the generation of many different kinds of graphs and drawing. Most PCs support multicolored graphic displays, and some allow certain types of graphics objects to be animated. The ability to generate colored animated displays, enhanced with sound effects, provides the foundation for the wide variety of computer games that have become so popular in recent years.

Most versions of BASIC include special graphics statements that allow a variety of graphic displays to be created easily. For example, instructions are available for generating common shapes, such as lines, dots, rectangle, circles, and ellipse. These shapes can be used to create a variety of sophisticated graphic displays that include the use of color and sound. Special statements are also available to generate animated displays. In addition the communication can be controlled by auxiliary input devices, such as mouse, joystick.

GRAPHICS FUNDAMENTALS

The fundamental elements of graphic display are small dots called pixels (pictures elements). These pixels can be combined to form more complex shapes just as characters are combined to form words, sentences, and paragraphs.

The level of detail (i.e the resolution) of a graphic display is measured in terms of the largest number of horizontal and vertical pixels that can be displayed at any one time. These values will be determined by the computer’s hardware. Personal computers typically support graphic display of 640 (horizontal) by 350 (vertical) or 640 by 480 pixels. These values will vary from one computer to another. Higher resolutions (e.g 1024 by 1024 or 96 by 4096) can be obtained with more expensive equipment.

The maximum available number of colors is also determined by the computer hardware. Most PCS support at least 16 colors and some support as 256 different colors.

Quick basic support several different graphic modes (i.e several different levels of resolution) each mode is specified via a SCREEN statement. Screen 0 refers to the text mode, while screen 9 refers to EGA graphics; this graphic provides a display of 640 (horizontal) by 350 (vertical) pixels in 16 different colors.

The EGA graphics mode is activated by the statement SCREEN 9. This statement must precede the generation of any graphic display, the use of this statement requires that the computer be equipped with an EGA compatible display adapter and an EGA compatible monitor.

The choice of colors is specified by the color statement. The COLOR statement can include two parameters, the first of which specify the foreground color and second specifies the background color. Border colors are not specified. Each parameter must be integer value between 0 and 15, thus providing a choice of 16 different colors.

COLORS AND THEIR CORRESPONDING NUMERICAL PARAMETERS

0 Black	**6 Brown**	**12 Light Red**
1 Blue	**7 White**	**13Light Magenta**
2 Green	**8 Gray**	**14 Yellow**
3 Cyan	**9 Light Blue**	**15 High intensity White**
4 Red	**10 Light Green**	
5 Magenta	**11 Light Cyan**	

Example 1

Generate an EGA graphic display in BASIC with black foreground and yellow as a background color.

Solution

```
Screen 9
Color 0, 14
```

POINTS AND LINES

PSET and **PRESET** are two statements that generate single points at any specified location on the screen and in any color.

PSET STATEMENT:- this statement can be used with a specified color, thus overriding the foreground color that is specified in the color.

To use **PSET** statement, the word **PSET** must be followed by a point of parameters, enclosed in parentheses and separated by a comma: e.g **PSET** (320,175). These parameters indicate the X and Y coordinates of the point.

The parameter after the two coordinates is optional which indicates the color of the point.

Example 2

```
Screen 9
Color 10,4
PSET (300,174),0
```

The first statement specifies EGA graphics and the second specifies light green foreground against red background. The third statement causes a black point to be generated at the centre of the screen.

Example 3

Write a program that generates 950 different points at random locations on the screen, using EGA graphics.

Solution

```
CLS
```

```
Screen 9
Color 15,0
Randomize Timer                          ‘initialize the random number generator
For I = 1 to 950

Let X = INT (640*RND)                  ‘generate a random X coordinate
Let y = INT (350 * RND)                ‘generate a random Y coordinate

PSET (X,Y),CLR                         ‘generate a random color
Next i
End
```

<u>PRESET STATEMENT</u>:-this is used to generate a single point. By default the background colour is automatically selected if the color parameter is not explicitly shown. It is possible in some application to use PSET to generate a set of points and PRESET to “erase” the points by generating the same points in the background colour.

Example 4

```
SCREEN 9
COLOR 15,1
PSET (320,175),14
FOR I = 1 TO 2000
NEXT I
PRESET (320,175)
```

Some versions of BASIC include LINE statement that allows a straight line to be drawn between any two points on the screen. Both points can be specified explicitly, or one of them can be the last point specified.

General Format

```
LINE (X,Y) –(X,Y)                       X and Y represents coordinates
```

<u>Example 5</u>

```
SCREEN 9
Color 15, 1
LINE (25,55) – (600, 250), 14
```

The first two statements specify EGA graphics with a high intensity white foreground and a blue background. The line statement generate a yellow diagonal line running from upper left (25,55), to lower right (600, 250). Note that the last parameter (14) overrides the foreground color in the COLOR statement, thus generating a yellow line rather than white.

<u>SHAPES</u>

BASIC includes statements that allow simple shapes to be drawn. For example quick BASIC includes a special form of the LINE statement that will generate a rectangle and a circle statement that can generate circles and ellipses. It is possible to filled the shapes with the colour.

Example 6

A program to draw the rectangle whose diagonal connects two points.

Solution

```
SCREE N 9
Color 15, 1
LINE (20, 50) – (620, 300), 14,B
```

EXPANDING RECTANGLE

A Program which causes a sequence of rectangle to move from the center of the screen to the outer edges. Group rectangles are generated in an alternating foreground colours, creating the illusion of "pulses" of rectangular shapes originating at the center of the screen.

Example 7

```
Rem *****EXPANDING RECTANGLES************
DEFINT A-Z
RANDOMIZE TIMER
SCREEN 9
CLS
LET clr = 15                    'initial foreground color
LET P =  0                       'draw rectangle size
LET Q = -60                       'erase rectangle size constant
Let delay = 2000                  'time delay constant
'MAIN LOOP
DO WHILE P>=0
LET X1 = 320 – 3*P:LET Y1 = 175- 1.7 * P
LET X2 = 320 + 3 * P: LET Y2 = 175 + 1.7 *P
LINE (X1,Y1) - (X2,Y2),CLR,B           'draw a rectangle
For I =1 to delay: next i
If Q>= 0 then
LET U1 = 320 – 3*Q : LET V1 = 175-1.7*Q
  LET U2 = 320 + 3*Q : LET V2 = 175+1.7*Q
LINE (U1,V1)- (U2,V2),0,B                      'erase a rectangle
For I = 1 to delay : Next I                          'time delay
ENDIF
LET P= P+5
IF P>100 THEN
LET P= 0                                              'reset the draw offset
LET  clr=1+INT(15 * RND)                                  'reset the colour
ENDIF
LET Q = Q+5
IF  Q > 100then let q=0                          'reset the erase offset
LOOP
end
```

example 7

a program to generate circle of 320 by 175 with high intensity white foreground color and blue background colour, centered at the middle of the screen with a radius of 80 pixels.

Solution

```
Screen 9
Color 15, 1
Circle (320,175),80,14
```

ANIMATIONS

Animations display is interesting and entertaining applications that provide a BASIC computer games and they are an important part of many educational and artistic programs.

In the previous example we have seen some simple animations that are not moving but delay for some time and erase. Here we will see animation that involves movement of filled objects.

SIMULATION OF A BOUNCING BALL

```
' ****** BOUNCING BALL***********
Defint A-Z
Randomize timer
Let delay = 5000
Let Flag = 1
Screen 9: Color 15,1:cls
LINE (0,0)-(639,349), ,B                    'boader
LINE (10,7)- (629,342), , B
PAINT (5,2)
LET X = 20+INT(600*RND)                           'define an initial position.
LET Y = 20 + INT (310 *RND)
LET DX= INT(21*RND)-10
LET DY = INT(21*RND)-10
'main loop
Do while flag > 0
Circle (x,y),12,1                           'erase the ball
Paint (X,Y), 1
```

```
Let x1 = x+dx                         ‘new coordinates
Let y1 = y+dy
IF x1 < 23 THEN
Let x1 = 23
Let dx = -dx
ELSEIF x1 > 616 THEN                       ‘hit the left wall
Let x1 = 616
Let dx = -dx
ENDIF
IF y1<18 THEN                              ‘hit the top
Let y1 = 18
Let dy = -dx
ELSEIF Y1>331 THEN                           ‘hit the bottom
Let y1 =331
Let dy = -dy
Endif
Circle (x1,x2), 12                         ‘redraw the ball
Paint (x1,y1)
Let x = x1
Let y = y1
For I = 1 to delay : next time delay
Loop
End
```