

# 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.

# **FUNCTIONS OF COMPUTER PROGRAM**

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 ***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

# TYPES OF LANGUAGE TRANSLATORS

- Each language needs its own translator.  
Generally, there are 3 types of language translators:
  1. Assembler.
  2. Interpreter.
  3. Compiler



# 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.
  2. It assigns memory locations to the names the programmer uses.
  3. It generates the machine code equivalent of the Assembly instructions.

# 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**.

# Functions of a compiler.

- A Compiler performs the following tasks during the compilation process:
- It identifies the proper order of processing, so as to execute the process as fast as possible & minimize the storage space required in memory.
- It allocates space in memory for the storage locations defined in the program to be executed.
- It reads each line of the source program & converts it into machine language.
- It checks for **Syntax errors** in a program (i.e., statements which do not conform to the

# Differences between Compilers and Interpreters

Interpreter	Compiler
<ol style="list-style-type: none"><li>1. Translates &amp; executes each statement of the source code one at a time.</li><li>2. Translates the program each time it is needed for execution; hence, it is slower than compiling.</li><li>3. Interpreted object codes take less memory compared to compiled programs.</li><li>4. For an Interpreter, the syntax (grammatical) errors are reported &amp; corrected before the execution can continue.</li><li>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.</li></ol>	<ol style="list-style-type: none"><li>1. Translates all the source code statements at once as a unit into their corresponding object codes, before the computer can execute them.</li><li>2. Compiled programs (object codes) can be saved on a storage media and run when required; hence executes faster than interpreted programs.</li><li>3. Compiled programs require more memory as their object files are larger.</li><li>4. For a Compiler, the syntax errors are reported &amp; corrected after the source code has been translated to its object code equivalent.</li><li>5. Once the source program has been translated, it is no longer available to the Compiler, so the error messages are usually less meaningful.</li></ol>

# 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:

- Computer program.
- Programming.
- 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?



# **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.

# 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:
  - i. Machine languages.
  - ii. Assembly languages.

# Features of low-level languages

- 1) They are machine hardware-oriented.
- 2) They are not portable, i.e., a program written for one computer cannot be installed and used on another computer of a different family.
- 3) They use Mnemonic codes.
- 4) They frequently use symbolic addresses.

# Machine languages (1<sup>st</sup> 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 *Cyril processors*. Therefore, before interpreting the meaning of a particular code, a programmer must know for which CPU the program was written.

# Assembly language (2<sup>nd</sup> 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.

# 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.
4. They are therefore suitable for writing *Operating system software & Game programs*, which require fast & efficient use of the CPU time.
5. They require less memory space.
6. Low-level languages are stable, i.e., they do not crash once written.



# Disadvantages of Low-level languages

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*.
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.

# 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.
- 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 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**
2. High-level language programs are short, and take shorter time to be translated.
3. They are easy to learn, 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. High-level language programs are self-documenting, i.e., the program statements displays the transparency of purpose making the verification of the program easy.
8. 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.
7. 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:
  1. Sequence
  2. Selection.
  3. Iteration (looping).



# Advantages of structured programming.

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

# Examples of Third generation programming languages include

- **BASIC** (**B**eginners **A**ll-purpose **S**ymbolic **I**nstructional **C**ode)
- **PASCAL**
- 
- **COBOL** (**C**ommon **B**usiness **O**riented **L**anguage)
- **FORTRAN** (**F**ormula **T**RANslator)
- **Ada**
- **C**

# 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:
  -
- To speed up the application-building process, thereby increasing the productivity of a programmer.
- To enable quick & easy amendments and alteration of programs.
- To reduce development & maintenance costs.
- 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.
- To allow non-professional end-users to develop their own solutions.
- To generate bug-free codes from high-level expressions of requirements.

# Advantages of fourth generation languages.

- They are user-based, and therefore, easy to learn & understand.
- 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.
- Very little training is required in order to develop & use 4GL programs.
- 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:

1. PROLOG

2. LISP

3. Mercury

4. OCCAM.

# 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.

- Examples of Object-oriented programming languages are: -
  1. Simula
  2. C++
  3. SmallTalk
  4. 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.

# 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*.

- Examples of 4GLs are:
  - visual Basic
  - Delphi Pascal
  - Visual COBOL (Object COBOL)
  - Access Basic

## **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
  6. Development time
  7. Popularity
  8. Documentation
  9. Maintenance
  10. Availability of skilled programmers

# Review Questions

- (a). What is a Programming language?
  - (b). Explain the two levels of programming languages.
  - (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.
  - (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.
  - (a). What is meant by *program portability*?
  - (b). Why are low-level languages not considered to be portable?
- 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.



- 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.

# **Problem definition (Problem Analysis).**

In Problem definition, the programmer tries to define (determine) the:

- Output expected from the program.
- Inputs needed to generate the output information.
- Processing activities (requirements), and
- 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.

# 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.

# 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 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.

# 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.

# Types of program errors

- There are 5 main types of errors that can be encountered when testing a program. These are:
  - Syntax errors.
  - Run-time (Execution) errors.
  - Logical (arithmetic) errors.
  - Semantic errors.
  - Lexicon errors

## 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:

Pseudocodes.

- Flowcharts.
- Decision Tables.
- Decision Trees.

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

# PSEUDOCODES

- 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.

- The statements must be short, clear and readable.
- The statements must not have more than one meaning (i.e., should not be ambiguous).
- The pseudocode lines should be clearly outlined and indented.
- 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.
- 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.*

# Example 1: *pseudocode*

START

PRINT “Enter two numbers”

INPUT X, Y

Sum =  $X + Y$

Average =  $\text{Sum}/2$

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.*

# Example 2: 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.*



## Example 3: PSEUDOCODE

START

Set  $\pi$  to 3.14

Prompt the user for the Radius (R)

Store the radius in a variable (R)

Set Diameter to  $2 * \text{Radius}$

Set Circumference to  $\pi * 2 * \text{Radius}$

Set Area to  $\pi * \text{Sqr}(\text{Radius})$

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
  - $E = M * C * C$
  - Display E
- STOP

# NB:

- It is important to use program control structures when writing Pseudocodes. The most common constructs are:
- **Looping (Repetition / Iteration)** – used where instructions are to be repeated under certain conditions.
- **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”.*

# Example 5:PSEUDOCODE

START

PRINT “Enter the Age”

INPUT Age

IF Age > 20 THEN

PRINT “Adult”

ELSE

PRINT “Young person”

STOP

# 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:
- **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.


- **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)***.
- **Terminal symbol./START 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.


- Input or Output symbol.

-  (*Parallelogram*)

- - It is used to identify/specify an input operation or output operation.
- For example;



- **Process symbol.**

-  (*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.

# Advantages of using Flowcharts

- Quicker understanding of relationships.
- Effective synthesis.
- Proper program documentation.
- Effective coding.
- Orderly debugging and testing of programs.
- Efficient program maintenance.

# Limitations of using Flowcharts.

- Flowcharts are complex, clumsy & become unclear, especially when the program logic is complex.
- If changes are to be made, the flowchart may require complete re-drawing.
- Reproduction of flowcharts is usually a problem, since the flowchart symbols cannot be typed.
- No uniform practice is followed for drawing flowcharts as it is used as an aid to the program.
- Sometimes, it becomes difficult to establish the link between various conditions, and the actions to be taken upon a particular condition.

# General guidelines for drawing a program flowchart

- A flowchart should have only one entry/starting point and one exit point
- The flowchart should be clear, neat and easy to follow.
- Use the correct symbol at each stage in the flowchart.
- The flowchart should not be open to more than one interpretation.
- Avoid overlapping the lines used to show the flow of logic as this can create confusion in the flowchart.
- Make comparison instructions simple, i.e., capable of YES/NO answers.
- 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.
- 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.
- Check to ensure that the flowchart is logically correct & complete.

# Revision Exercise.

- Define the following:
  - Algorithm.
  - Pseudocode.
  - Flowchart.
- **(a)**. State the various types of flowcharts.
- **(b)**. Discuss the advantages and disadvantages of flowcharts.

# 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.

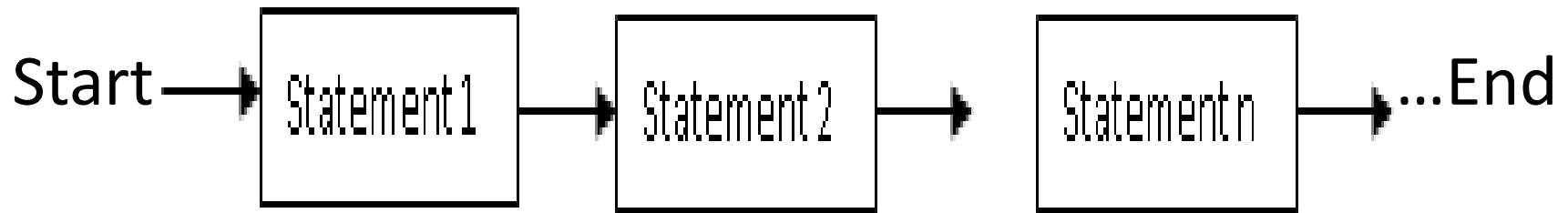


# Types of control structures

- There are 3 control structures used in most of the structured programming languages:
- Sequence.
- Selection.
- 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***.



- **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.,  $X \geq 20$
- 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.

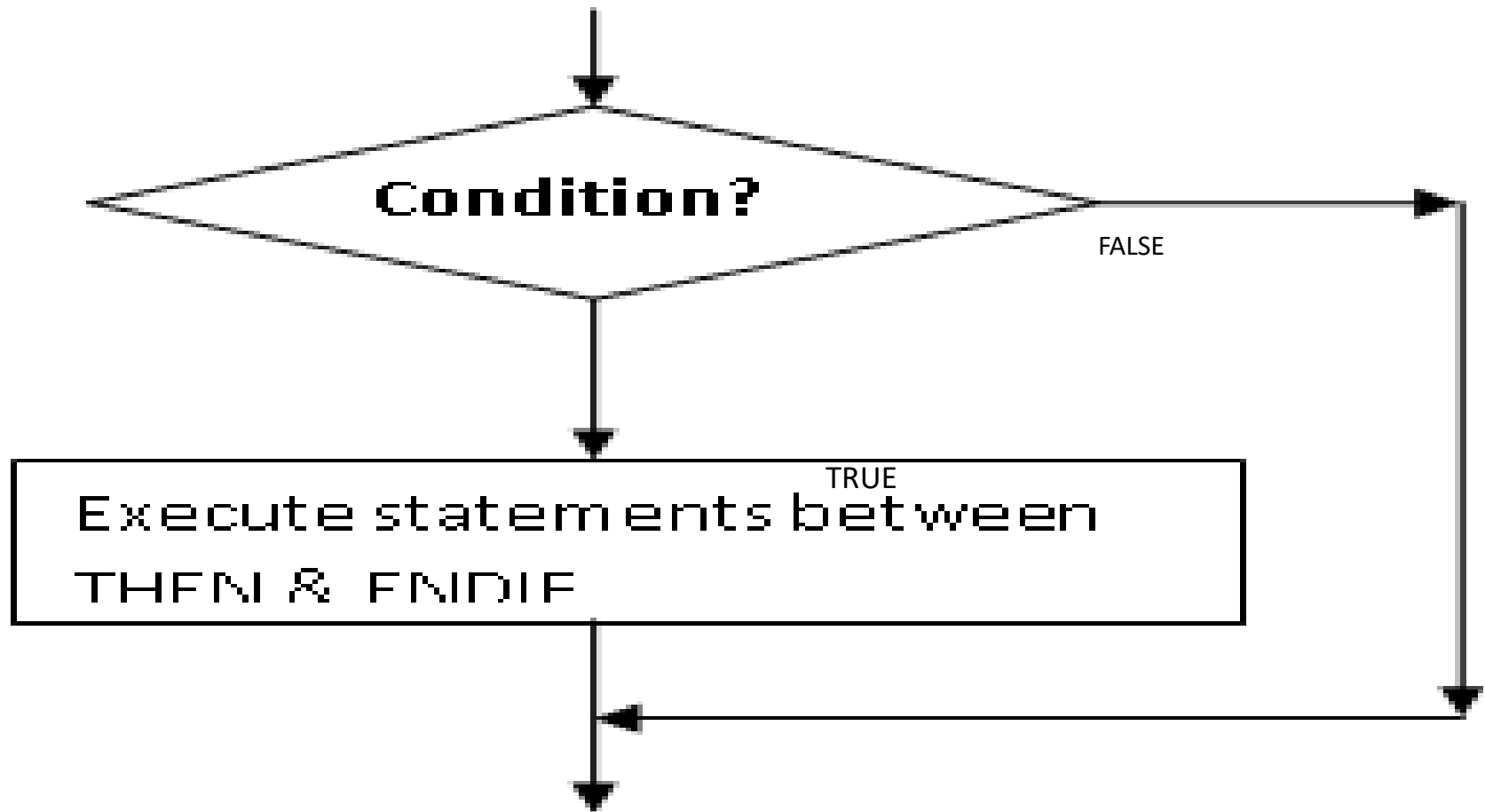
# Types of selection control structures

- 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

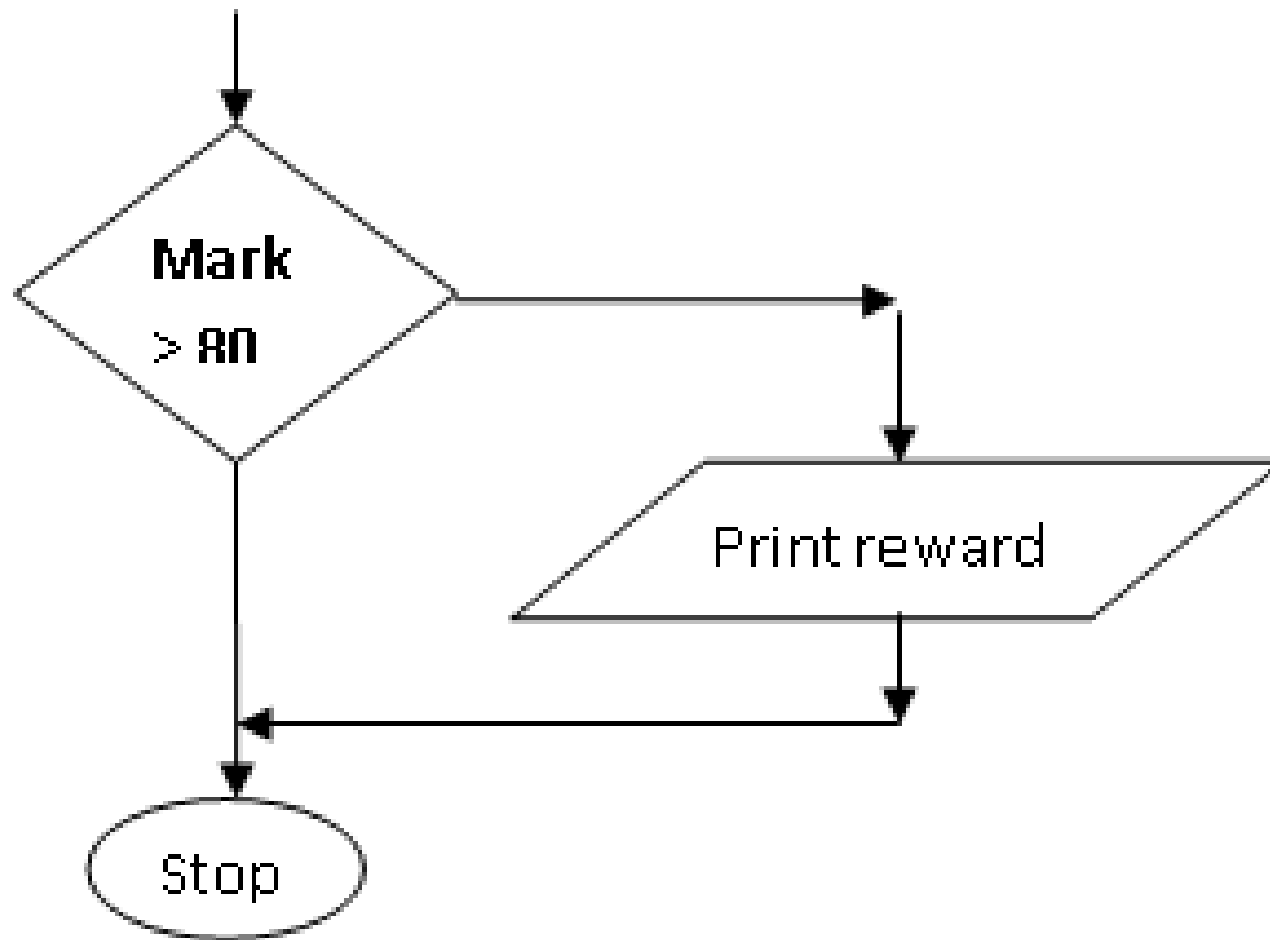
# IF – THEN

- 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

# The diagrammatic expression of the IF-THEN structure



# Example



# 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)

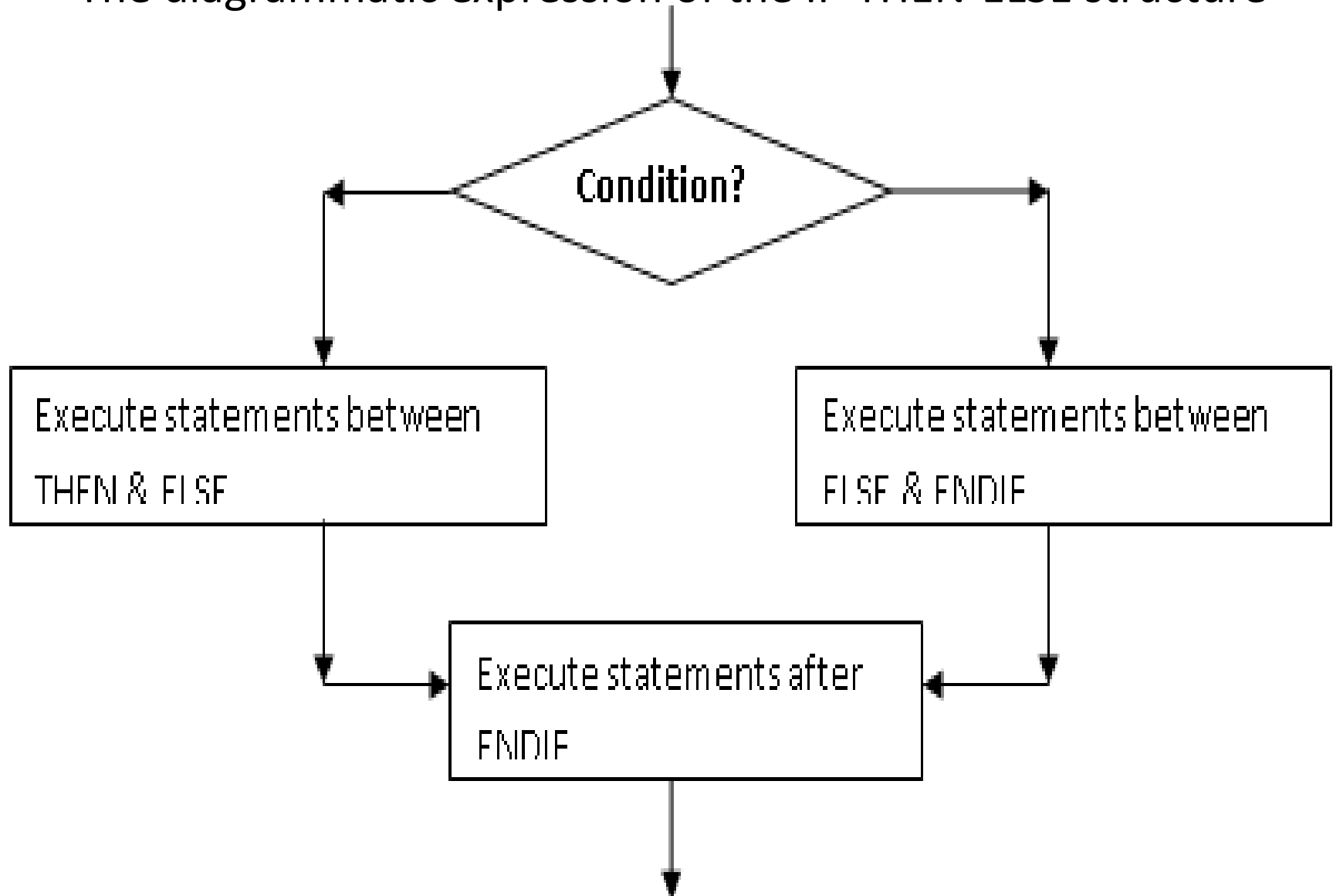
ELSE

*Statement 2;* (called the ELSE part)

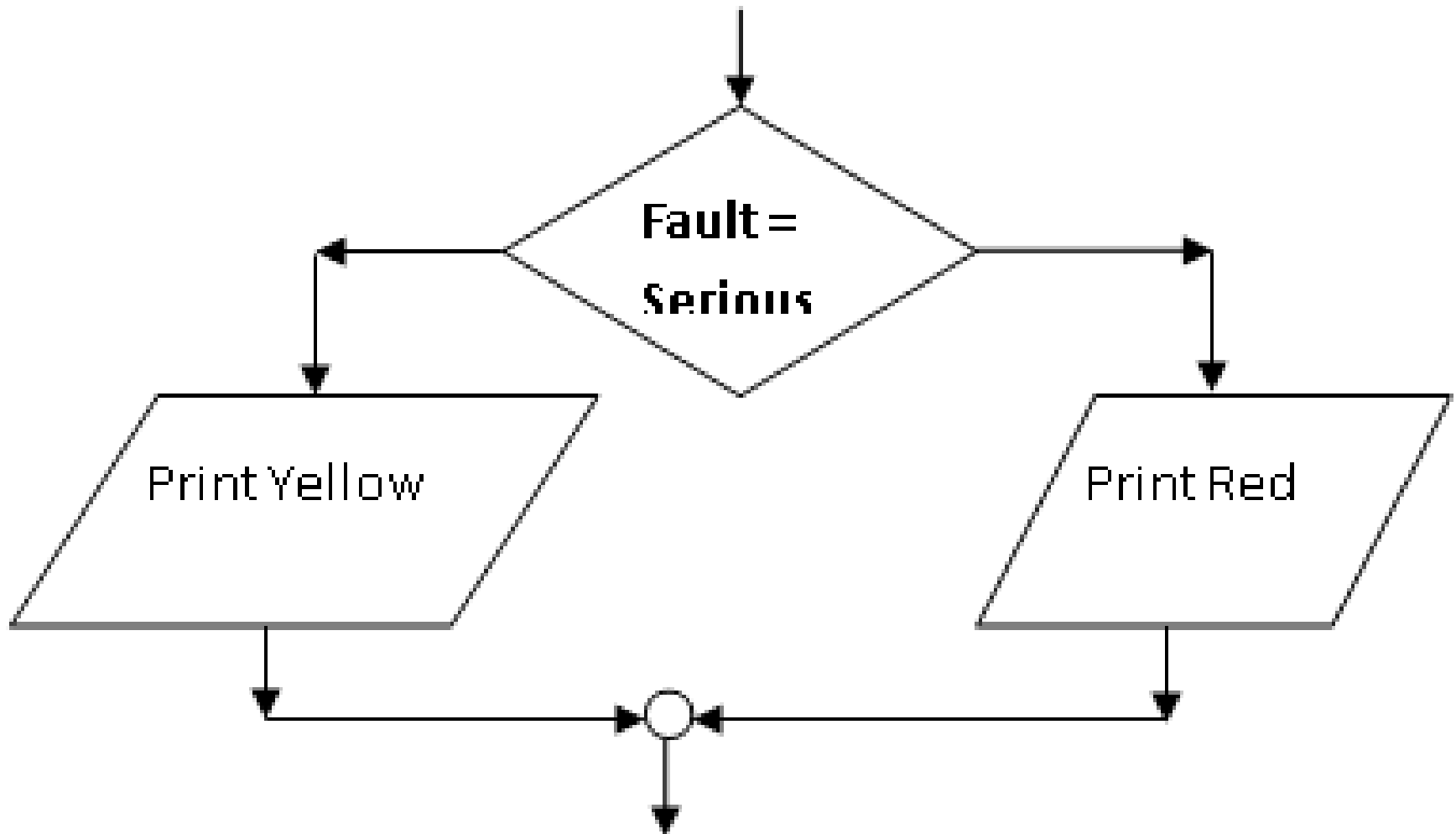
ENDIF (indicates the end of the  
control structure)



The diagrammatic expression of the IF-THEN-ELSE structure



# Example: Flowchart



# Example: Pseudocode

IF Fault = Serious THEN

    Print “ Give red card”

ELSE

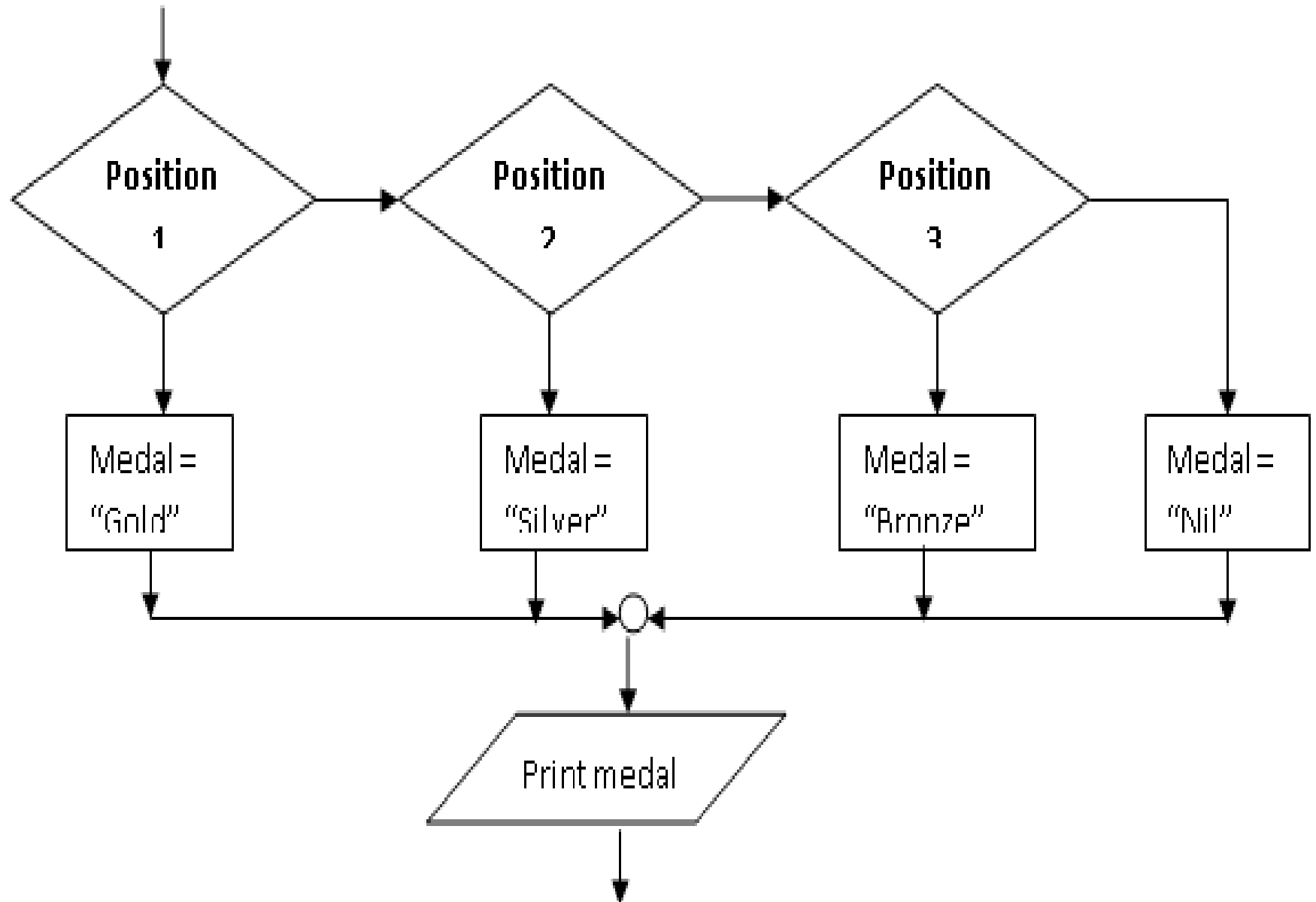
    Print “ Give Yellow card”

ENDIF

# 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
- ENDIF

# Flowchart.



# 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

.

.

*Label n: statement n*

*ELSE*

*Statement m*

*ENDCASE*

# Example: 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');

ENDCASE

STOP

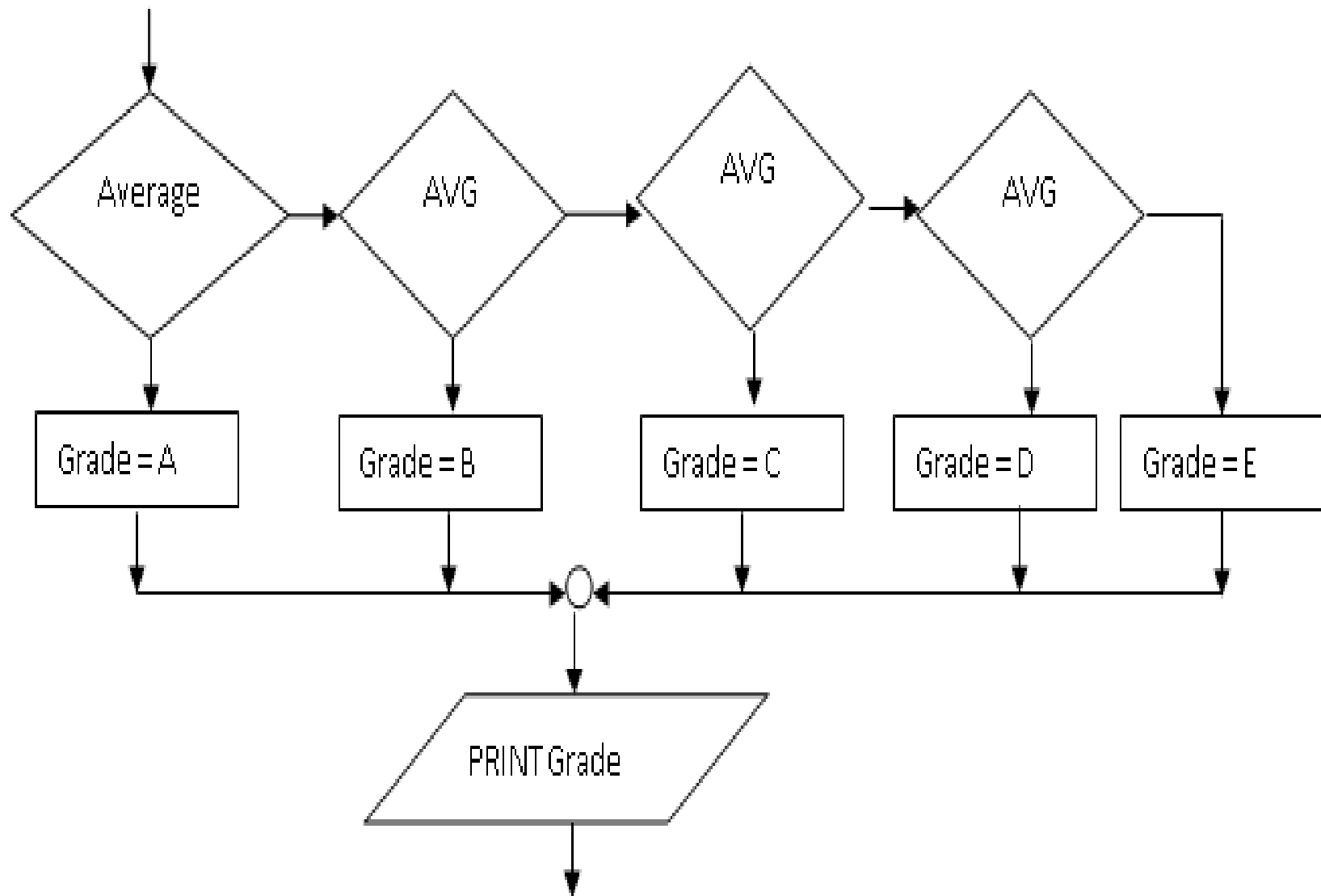
- The CASE structure consists of:
- The word CASE.
- A Control variable (e.g., **Day**).
- The word OF.
- A group of one or more statements, each group labeled by one or more possible values of the control variable.
- The word ENDCASE, indicating the end of the construct.



## ***Example : 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



# 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:
- **Loop body**, which represents the statements to be repeated.
- **Loop control**, which specifies the number of times the loop body is to be repeated.

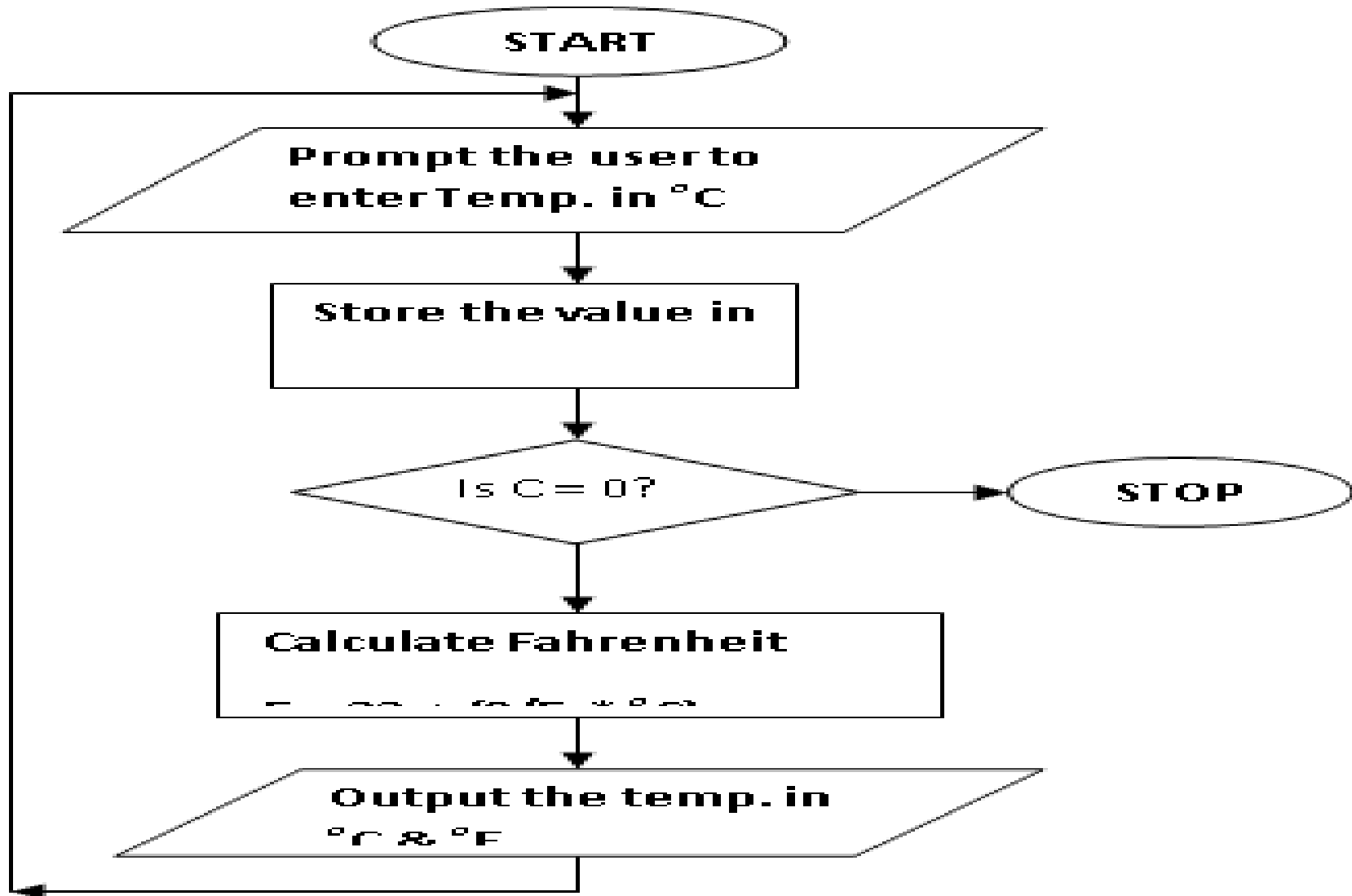
# ***Types of loops***

- ***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 + (°C \* 9/5)  
**STEP 5:** [Output temperature in °C & °F]  
**STEP 6:** [GOTO Step 1]

# **Example: Flowchart**



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

- ***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]
- $$F = 32 + (°C * 9/5)$$
- **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:***

- *Control variable (Counter)*: - it tells/instructs the program to execute a set of statements a number of times.
- *Initialization*: - allocating memory space, which will be occupied by the output.
- *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

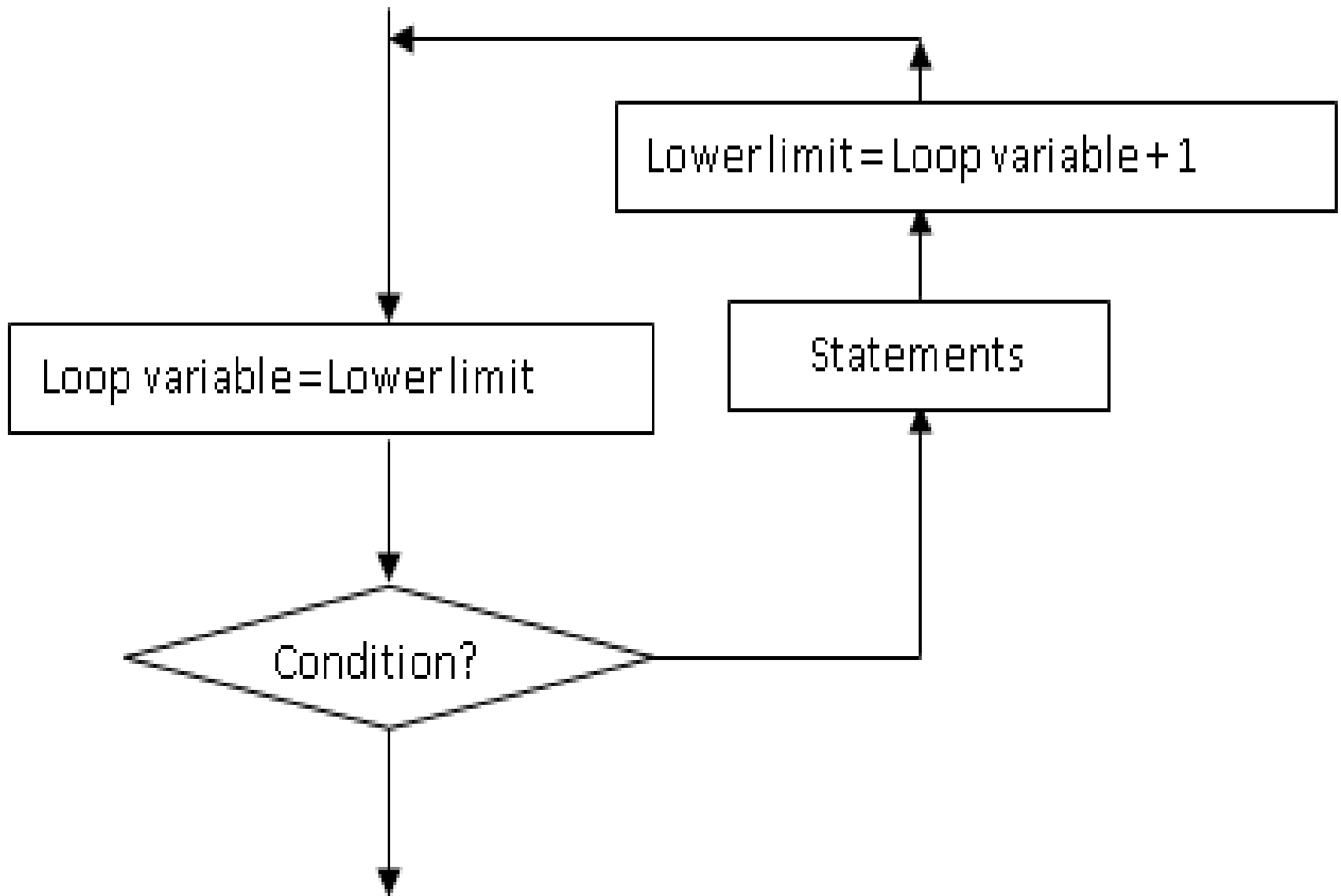
- The FOR loop is used in situations where execution of the chosen statements has to be repeated a predetermined number of times.
- The general format of the FOR loop is:

*FOR loop variable = Lower limit TO Upper limit DO*

*Statements;*

*END FOR*

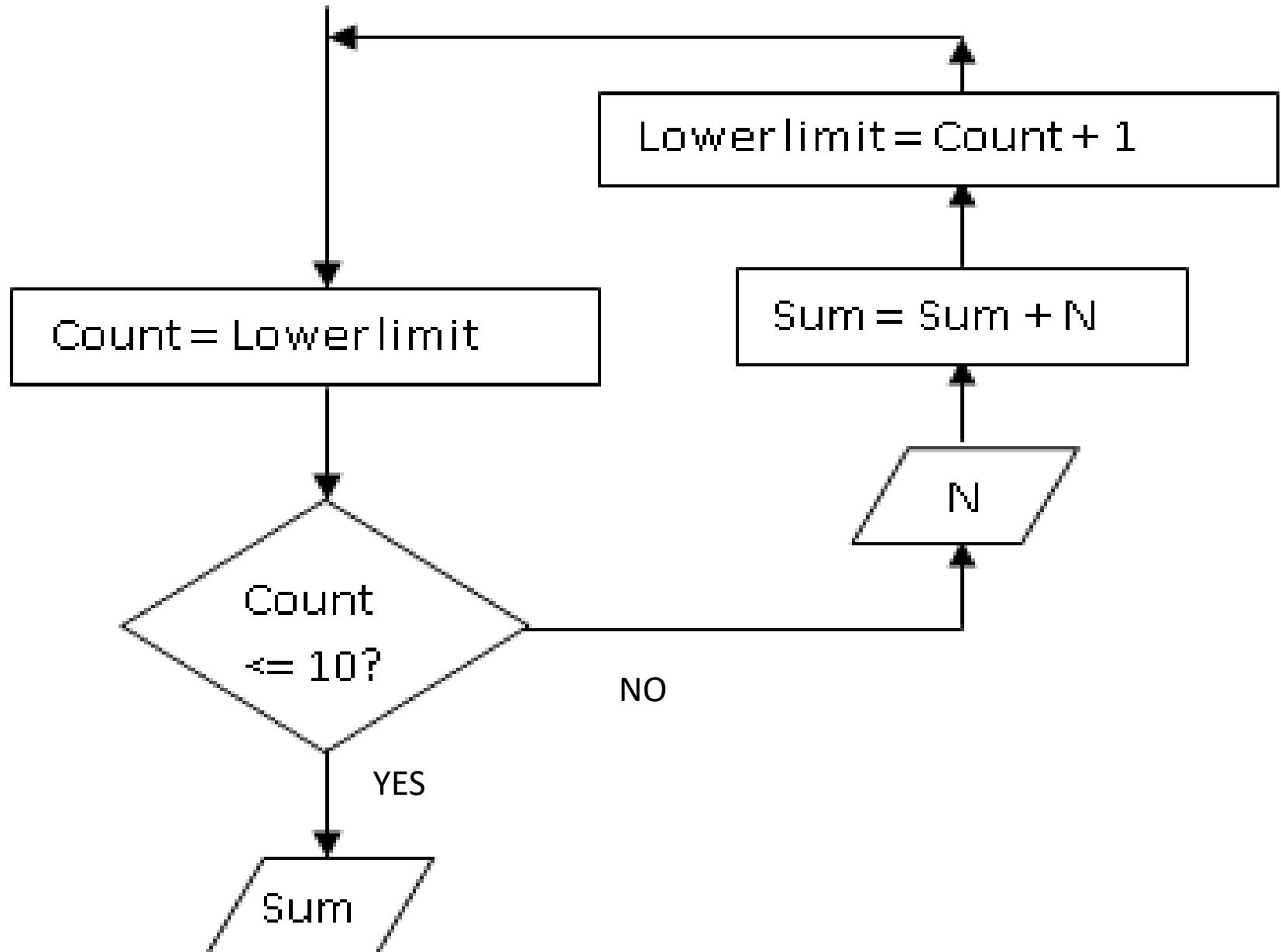
The flowchart extract for a FOR loop that counts upwards is:



# Example: Pseudocode

```
FOR count = 1 TO 10 DO  
  PRINT "Enter a number (N)"  
    Sum = Sum + N.0  
END FOR  
Display SUM
```

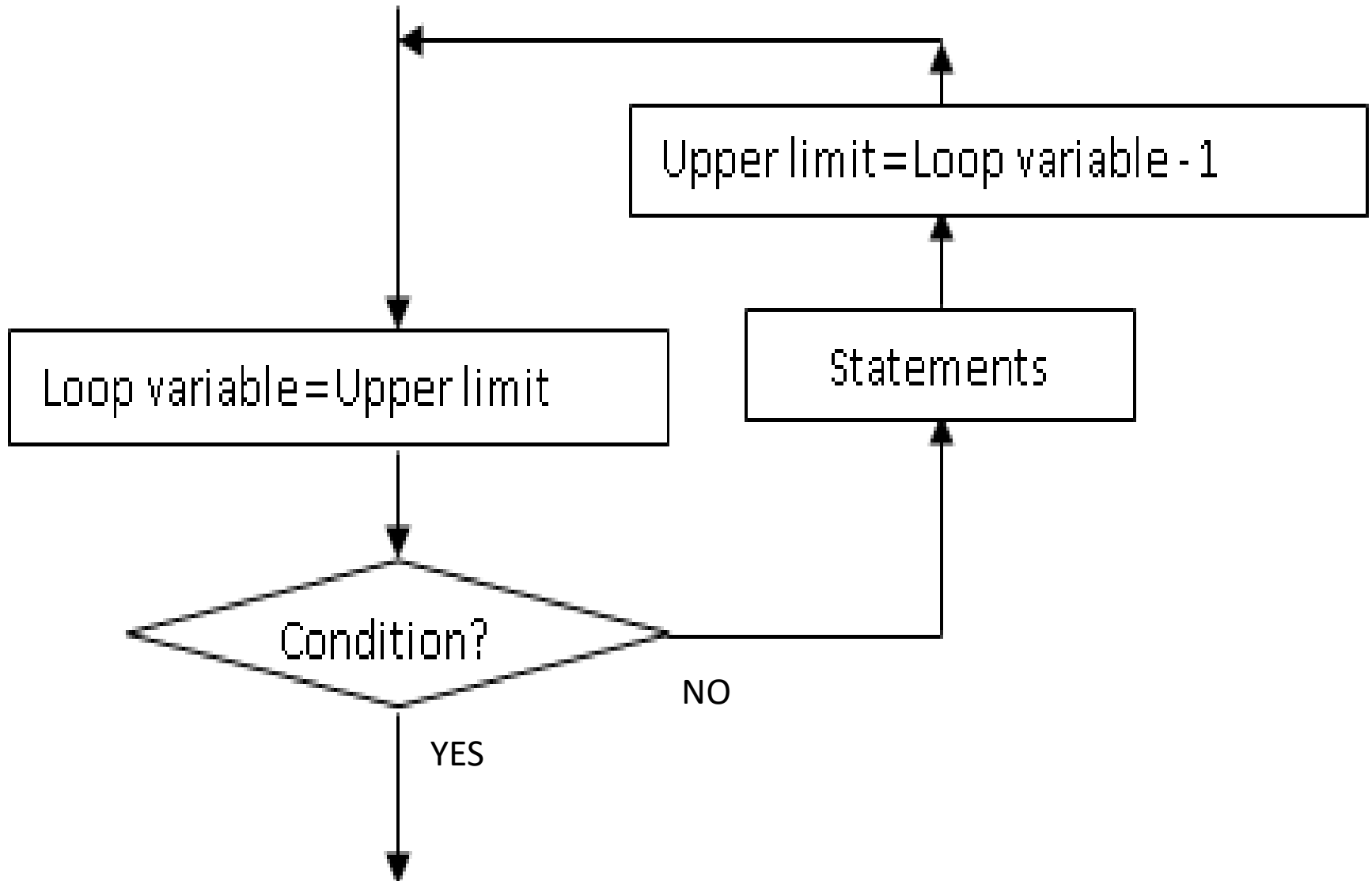
# Flowchart



***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:



# 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.

# The general representation of the WHILE loop is:

- **Pseudocode segment**

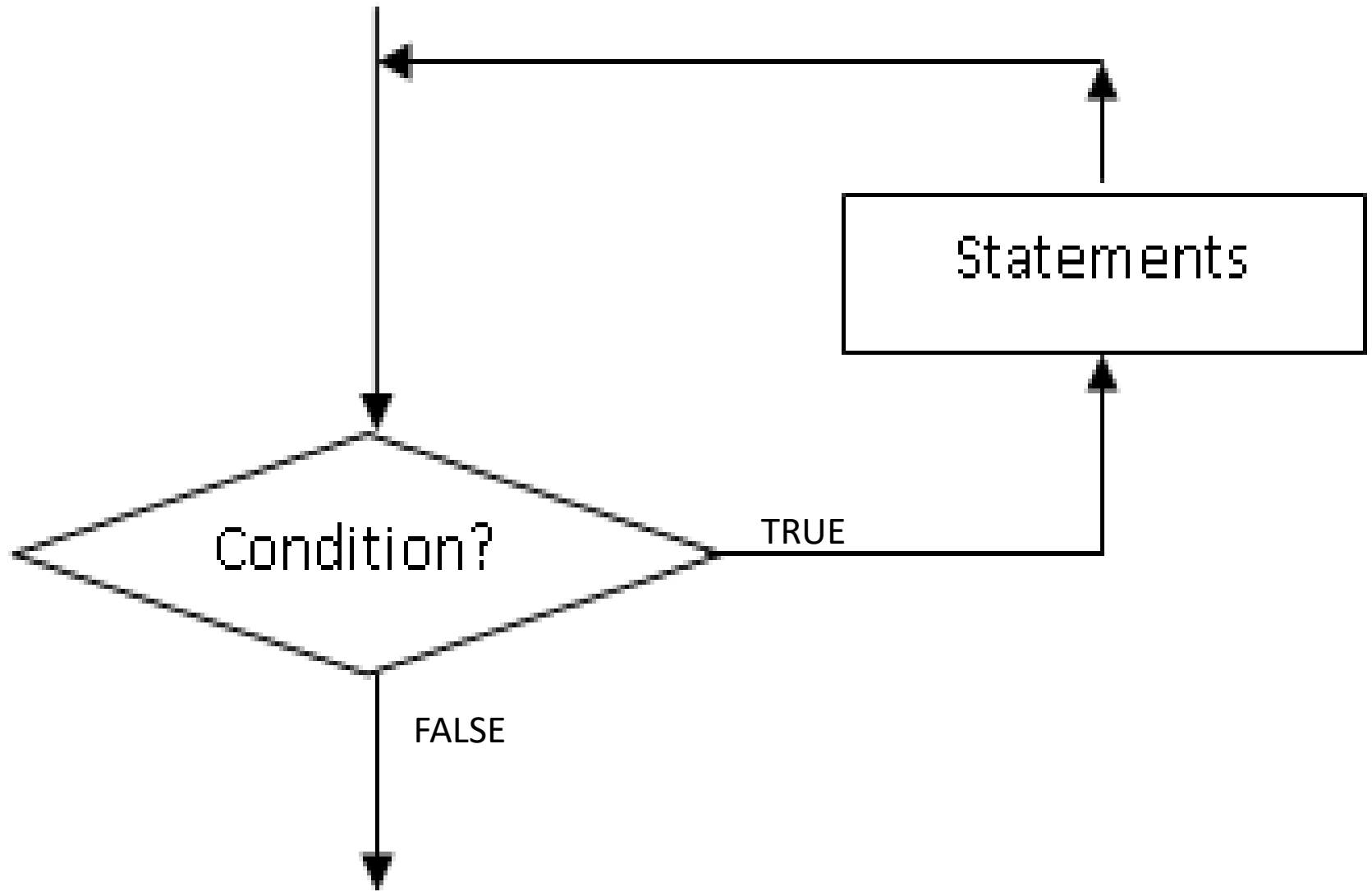
WHILE *Condition* DO

*Statements;*

ENDWHILE



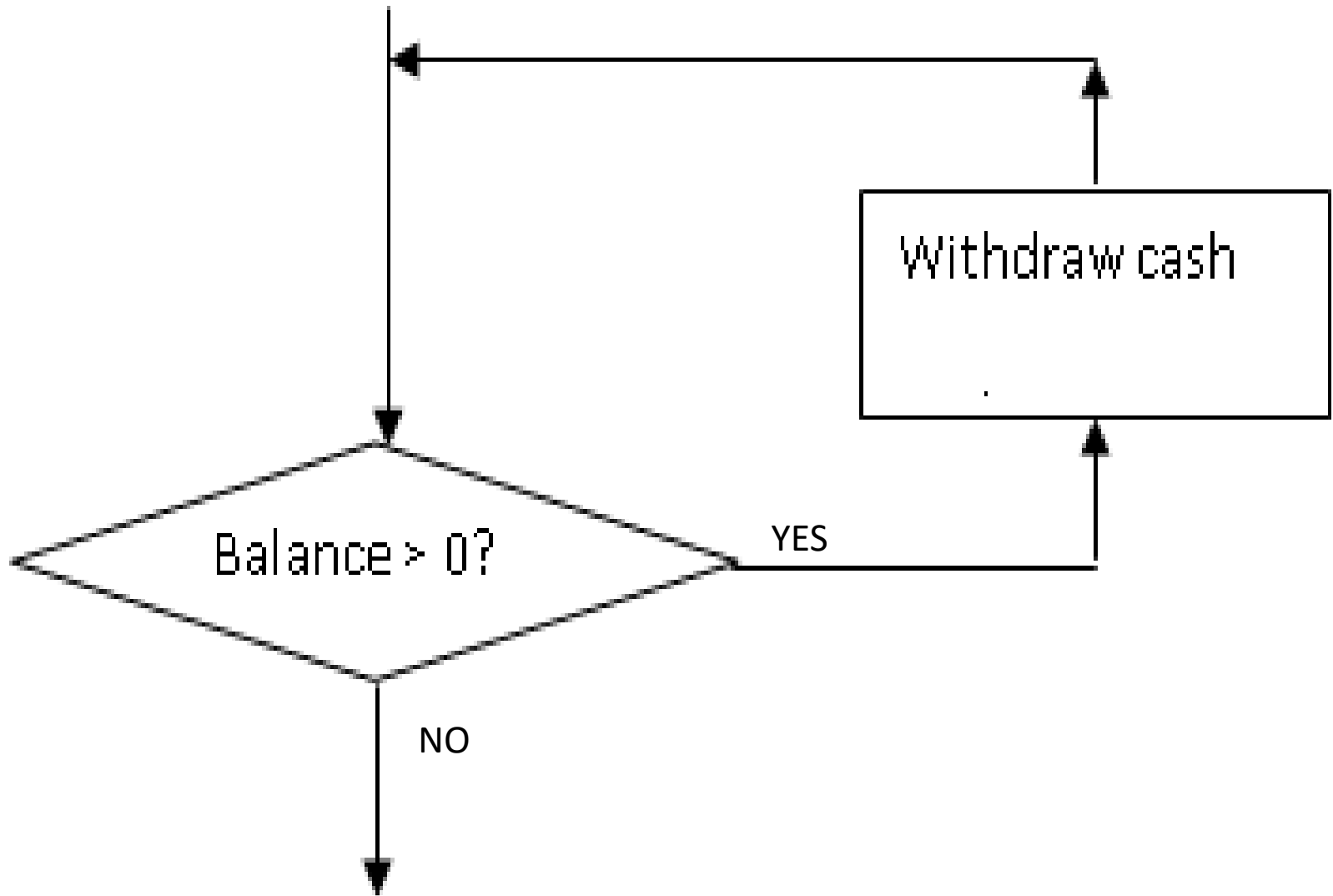
# Flowchart extract



# Pseudocode

```
WHILE Balance > 0 DO  
    Withdraw cash  
    Update account  
ENDWHILE
```

# Flowchart



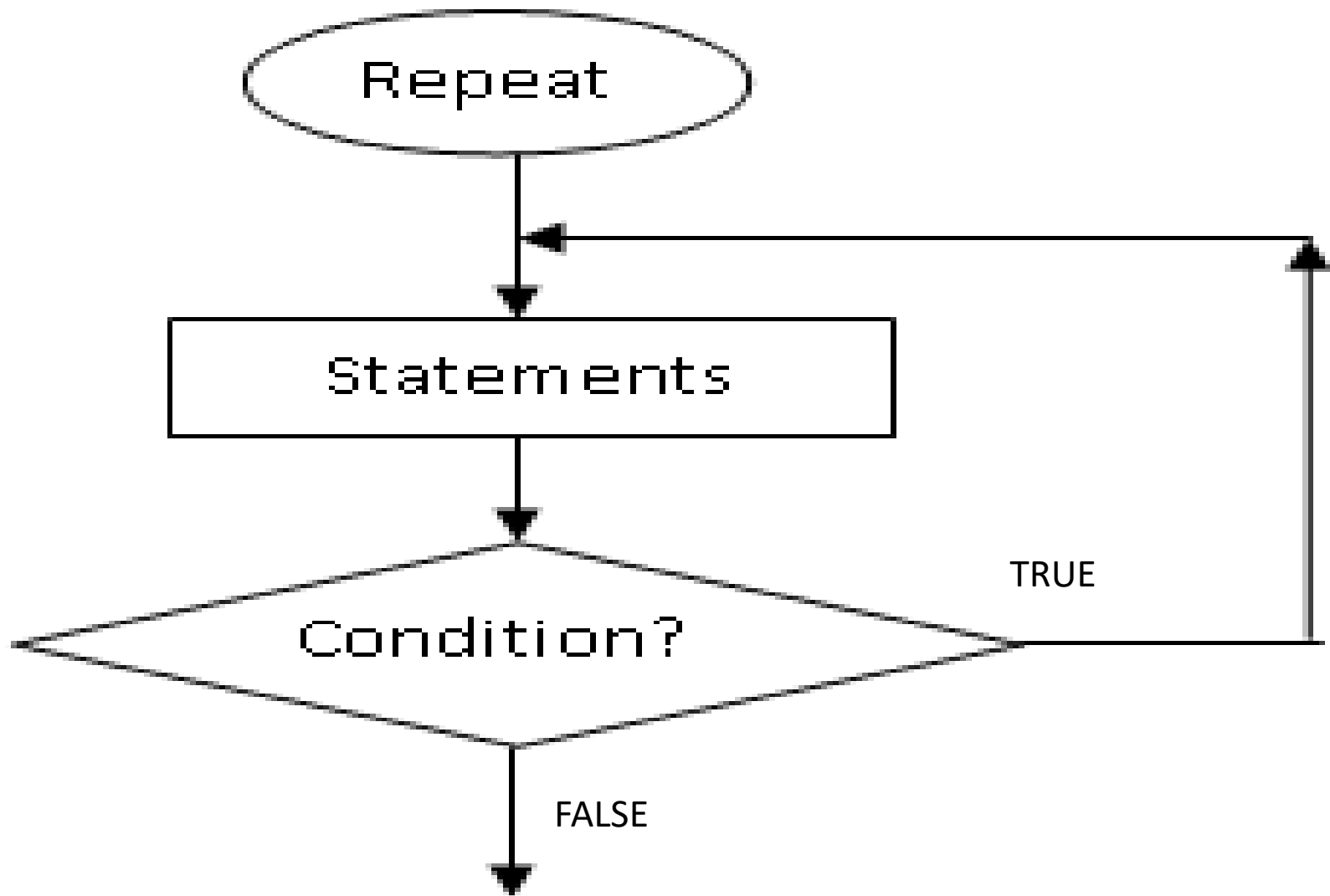
# The REPEAT...UNTIL loop

- 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.

# The general format of the REPEAT...UNTIL loop is:

- **Pseudocode segment**
- REPEAT
- *Statements;*
- UNTIL *Condition*

# Flowchart extract



- REPEAT
- Withdraw cash
- 
- Update account
- UNTIL balance  $\leq 0$ ;

# Flowchart

