CHAPTER ONE

CONCEPTS AND PRINCIPLES OF A PROGRAMMING LANGUAGE

LEARNING OBJECTIVES:

At the end of reading this chapter, the reader should gain the following.

* Understanding of the architecture of a microcomputer system.

* Understanding of Machine and Assembly language programming concepts.

* Understanding and appreciation of the need for high level programming languages.

* A fundamental understanding of compilers, loaders, linkers, and other high level language translators. The stages during compilation of a high-level source program to machine or object code.

* Understanding of the manner in which statements written in a high level source program are evaluated during compilation and the use of binary trees to construct evaluation structures.

* General understanding of data types, data structures and how they are manipulated on computer systems.

INTRODUCTION

A computer is any device or machine which operation can be ordered under the control of a program to solve a specific problem. There are many different kinds of computing devices to be found at various points in human civilisation and more recently, during the industrial revolution. By a computer program we mean the configuration of a device or machine that affects the way the device or machine operates or behaves under the influence of some input energy source. Therefore, there are permanently programmed systems such as those found in industrial machines and there are re-programmable devices most of which are electronic systems. I have given broad definitions to the terms "Computer" and "Computer Program" in preceding paragraph. However, in this manual we confine our interest to digital computer systems that operates on the principle of the stored program concept. Therefore, a more concise definition of a Computer Program is as follows.

A computer program is a predetermine sequence of instructions that specify the operations to be done by a computer on data values. An instruction is a sequence of Binary Digits (bits) that can be used to control the internal electronics of a digital computer system. Therefore, instructions are machine codes that specify what operation is to be performed by the electronics of the computer system. On a digital computer, programs at machine level consist of a series of binary words that are directly interpreted by the central processing unit (CPU). These specific sequences of instructions are designed to solve a particular problem on a selected device. Programming is the process of selecting the correct sequence of machine instructions to be used for solving a specific problem. The binary coded instructions used on a selected electronic device (computer system) are determine by the electronic configuration of the device. However, there are fundamental concepts and principles that govern the way programs are written and how they are executed. Writing machine programs require that you have knowledge of the electronic configuration of the machine. In addition, a program written for a given machine is not portable to other machines. Therefore, writing machine programs is best left to people who like to do things using 1's and 0's.

We invent assembly language as an alternative to machine language. Assembly language is machine language written using mnemonic. Like machine language, assembly language is also specific to the configuration of a given machine and therefore requires knowledge of the machine architecture. However, instructions are written using mnemonics and the Hexadecimal number system.

Like BASIC, assembly language consists of a set of commands that can be written into a program that tells the computer system what to do. However, each instruction in the assembly language instruction set refers directly to the digital components of the computer system. From this we can see that assembly language provides an interface between the programmer and the electronic configuration of the computer system. However, this interface is limited in its effect. It does not provide full abstraction of the internal complexity of the computer system. High level programming languages provides a user interface that abstract both the format in which instructions are executed by the system and the electronic complexity of the system on which a program is to be executed.

From this we see that a programming language forms an interface between the programmer and the hardware system of the computer. There are levels of abstraction between the hardware system and the programming language. These abstractions are logical and are built on consistent rules that manifest themselves at the software interface as syntax and semantic rules in particular programming languages. Good programmers have a comprehensive knowledge of the way digital computer systems works and is able to appreciate the syntax and semantic rules in well defined programming languages. Understanding of a computer system includes basic knowledge of the way the hardware system works at machine or assembly language level.

Understanding of systems software such as programming languages includes basic knowledge of the construction and workings of compilers and other language translators. Therefore, we begin this chapter with a general explanation of fundamental concepts and principles on how the hardware and software systems of a computer work. It is hoped that this will help you the reader to better appreciate the various elements of restrictions and control in the use of various resources on a digital computer system.

PROGRAMMING LANGUAGES

A programmer uses a programming language to write coded instructions that can direct and control the operations of a computer. At machine level, the computer executes instructions written using 1‰and 0‰ A computer program is a series of instructions, which specify the operations to be done by a computer on data values. On a digital computer, programs at machine level consist of a series of binary words that are directly interpreted by the Central Processing Unit (CPU). These specific sequences of instructions are designed to solve a particular problem.

MACHINE LANGUAGES

Machine language is written using 1‰and 0‰ These two numeric characters are used for representing numbers in the binary number system. They are also used for representing data on a digital computer system because the basic element that make up and cause the computer system to work are transistors. Transistors are bistable devices that can be in a switched **on** or switched **off** state. Therefore, a series of transistors can be logically manipulated by using binary words. For example, the binary word $\mathbf{11011001_2}$ could be used to switch **on** or **off** a special circuit in the electronic system of the computer. Hence, machine language is written using binary words. Machine language is also conveniently written using the hexadecimal number system (base 16). The machine language used by a computer system is determined by the architecture of the computer system. Therefore, programming at machine level requires the programmer to remember the machine interpretation of long strings of binary words or hexadecimal numbers. The programmer is also required to have an understanding of the architecture of the computer system on which the machine program is to be executed. Therefore, programming at machine level tends to be tedious and is best left to digital electronic engineers or people who love to count or do things with 1‰and 0‰

Not many people will want to write programs for themselves using machine language. Therefore Assembly Language was invented so that more programmers will find it easier to use a computer system to solve their programming problems. Assembly language is machine language written using mnemonics. A mnemonic is a word or combinations of letters that suggest the meaning of the instruction it represent. Like BASIC, an assembly language consists of a set of commands that can be written into instructions that tells the computer what to do. However, each mnemonic word in an Assembly language instruction set refers directly to the digital components of the computer system. There are different kinds of digital computer systems. A system in which all logical and arithmetic operations are performed by a single electronic unit is called a microprocessor. Figure 1-1 illustrates the basic component of an example microprocessor system.

The instruction set of a microprocessor is the set of machine instructions that can be carried out by the microprocessor. A program written in Assembly Language gives the microprocessor detail electronic commands such as **ı input value from port 04ı ,ı add 7 to this valueı** , and **ı output result value to port 2ˆ**. Table 1-1 is a program written in machine code along with the Hexadecimal (HEX) equivalent of each binary word. The same program is also listed in Assembly Language. Note that

the actual content of a sequence of memory locations constitute the machine instructions. This machine instruction can also be written in mnemonic format to create the assembly code.

Table 1-1An example Machine/Assembly language program.

MACHINE CODE				ASSEMBLY CODE	
Memory Address		Memory Content		Op. Code	Operand
Binary	HEX	Binary	HEX	Mnemonic	HEX
0000000000000000	0000	11011011	DB	IN	
0000000000000001	0001	00000100	04		04
0000000000000010	0002	11000110	C6	ADI	
0000000000000011	0003	00000111	07		07
0000000000000100	0004	11010011	D3	OUT	
0000000000000101	0005	00000010	02		02

We use the DEC PDP-11 architecture to focus our explanation of machine and assembly language programming.

Machine instructions written in the binary number system are very tedious for human understanding. For example, in the PDP-11 architecture, the binary number below is a machine instruction which when executed causes the data value stored in register $\mathbf{1_{10}}$ $\mathbf{= 000001_2}$ to be moved into register $\mathbf{2_{10}= 0000010_2}$. **0001 | 000001 | 000010**

The CPU in the computer handles data values with word length of 4, 8 or 16 bits depending on the particular architecture of the CPU used by the manufacturer. A **microprocessor** is the CPU of a microcomputer and is usually a single large-scale integrated circuit (LSI). However, there are some minicomputers today that are constructed from one or more large-scale integrated circuits. The PDP-11 is made by Digital Equipment Co-operation (DEC) and is currently the most popular minicomputer system. Figure 1-1 shows the architecture of one microprocessor used by the PDP-11.

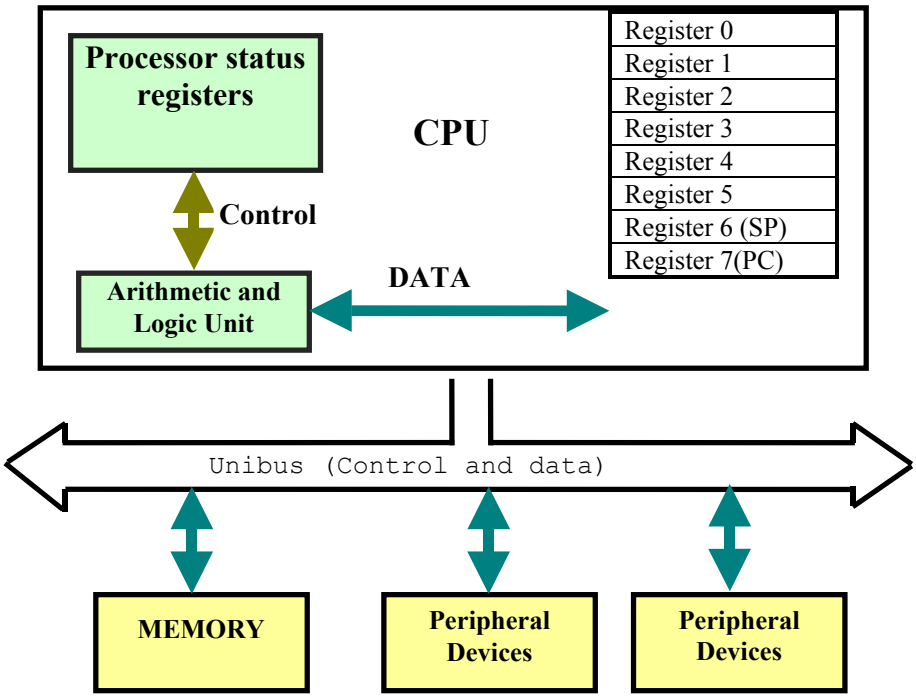

Figure 1-1
Architecture of the PDP-11 CPU. Register 6 is used as stack a pointer (SP) and register 7 is used as the program counter (PC).

The architecture illustrated in figure 1-1 is used to construct a 16-bit minicomputer. It contains 8 general-purpose registers, numbered 0 trough 7.

ASSEMBLY LANGUAGE

ı Because machine instructions are difficult to remember in binary, we have invented a way to represent each machine instruction in more English like manner and at the same time to convey or give a hint of the meaning of each instruction. In this representation, each instruction is given a **Mnemonic** name. This way of representing machine instruction is called **Assembly Language.** For example, the binary instruction: **0001 000001 000010** would be written in Assembly Language as **MOV %1 %2.** Another example of machine code written using mnemonic is **0 000 10 000 000100 → CLR %4** which means ı Clear register 4ˆ. The first example instruction **MOV %1 %2** consists of three parts **MOV, %1**, and **%2**. The first part of the instruction tells the computer what to do with the next two parts and is called the **Operational Code** or **OP CODE** for short. The next two parts of the instruction specify the location of the data to be used by the **OP CODE**. The next two parts of the instruction is called the **OPERAND.** The 2nd example instruction consists of two parts: **OPCODE OPERAND.**

The first instruction is in the format: **OPCODE OPERAND OPERAND .** This is the 2-address format while the 2nd example is the 1 address format. If a programmer writes a sequence of instructions using mnemonic, these sequence of instructions (a program) would first have to be translated (converted) to machine instruction (1‰and

0%) before they can be executed by a computer as meaningful machine commands. The machine code written using mnemonic is called an Assembly Language program and the special software that translates the assembly program to machine code is called an **ASSEMBLER.** Hence we see that, for a programmer to write programs in assembly language, the programmer must be familiar with the set of instructions that a particular machine can execute, i.e. the programmer must be familiar with the architecture of the machine on which the program is to be executed. An example program in assembly language is given below in table 1-2. The program adds three numbers together and then store the result in register number 3. The instruction **ADD X, Y** means; add content at location **X** to content at location **Y** and store the sum in location **Y**.

Table 1-2 Assembly language program

OP CODE	OPERAND	Comment
CLR	**%3 ;**	Clear register 3
ADD	**%0, %3;**	Add value at 0 to value at 3 (a)
ADD	**%1, %3;**	Add value at 1 to value at 3 (b)
ADD	**%2, %3;**	Add value at 2 to value at 3 (b)
HLT	**;**	Stop

To see how inadequate assembly language programming is, consider the following assembly program written on a different machine. In this example, the memory location for each machine instruction is given in the listed code.

Table 1-3 Assembly language containing bugs.

LOCATION	CONTENT	COMMENT
start ➔ 0	LDX 7	Load index register with value at 7
1	LDA 7,X	Load accumulator
2	DSZ	Decrement x and skip if zero
3	JMP 6	Jump to location 6
4	ADA 10,x	Add value at location 10, indexed to accumulator
5	JMP 2	Jump to location 2
end ➔ 6	HLT	Stop
7	4	Locations 7, 10-13 store data values used
10	16	by the program
11	32	
12	176	
13	24	

The intent of this program is to add the values of the four numbers in locations 10-13. However, the program will not work as intended because there are several errors in the coding. The correct program code is given in table 1-4.

HIGH LEVEL PROGRAMMING LANGUAGES

Assembly language was invented to offset the tedium of writing machine language programs. However, the use of assembly language still require the programmer to be familiar with the organisation of the hardware on which the program is to be executed. What we need is a system for writing programs that abstract the complexity of the internal electronics of the computer system. As you can see from the above examples, machine and assembly languages are very tedious to use. Therefore, we have invented programming languages in which the programmer is able to focus more on the actual problem to be programmed. The programmer is also able to use command statements that are within the problem domain as opposed to those machine instructions. We call these programming languages high level languages. In a high level programming language, the computer is given instructions using statements and expressions similar to those used in every day natural language. All high level language has rules that govern the way they may be used. The grammatical rules that govern the way statements and expressions are written in a language is called the **syntax** of the language. The rule that governs the way a program written in a language may be interpreted is called the **semantics** of the language.

Apart from solving the problems of tedium that we have identified above, there are several other advantages gained from the use of high level programming languages: High level languages are usually machine independent and are more problem specific. I.e., their syntax and semantics allow the programmer to write instructions that are more related to the problem to be solved rather than to the architecture of the computer on which the instructions are to be executed. High level languages are usually **portable**, i.e., programs written in these languages can be executed on a wide variety of machines with very little or no change. As a consequence of portability, languages must be standardised. Therefore, it is necessary that standards be established for programming languages. There are a number of international influential bodies that authorises these standards. The most influential is the **A**merican **N**ational **S**tandard **I**nstitute (**ANSI)**. ANSI have developed and adopted widely used standards for languages such as FORTRAN, Pascal, BASIC and COBOL. There is also the International Standard Organisation (**ISO**) that works with ANSI and many other national standardising institutions in other countries to set standards.

Most high level languages allow large programming problems to be easily broken down into a set of clearly defined modules, which would not be possible with assembly language. Therefore, it is much easier to do top-down programming with most high level languages. A unique feature of high level languages is their control structures. The use of certain control structures restricts the programmer from making logic errors in the design and coding of programs. This is because these control structures usually have fundamental logic that the programmer uses to implement a variety of program logic requirements. These special control structures convey more precisely the intent of a program than the primitive assembly language instructions does. For example, the equivalent of the previous assembly language program when written in the high level language **Algol** is as follows:

```
INTEGER ARRAY A(4) := (16, 32, 176, 24);
INTEGER  := 0
FOR  := 0 TO 3 DO SUM  := SUM + A(I);
```

Figure 1-2
The equivalent program to the assembly program illustrated in table 1-3 written in the high level programming language Algol.

Note that variables used in the above code are precisely declared and are separate from the algorithm that operate on them. **APL** is another language that would implement the above program objective in a more concise manner as:

```
SUM ← +/(16  32  176  24)
```

TRANSLATORS

A translator converts a **source** program into an **object** or **target** program. The source program is written in a source language. The object program belongs to an object language. The source language can be any assembly or any high level programming language. An assembler translates source code written in assembly language to object code in machine language. Assembly language is machine language written using mnemonics. There is a close relationship between assembly and machine language. Therefore, the relationship between assembly and machine code is simple and straightforward. An **interpreter** translates source code along with the source data at the same time one line at a time. Translation of the internal source code to machine code takes place at run time and no overall single program object code is created. The equivalent machine code for each line in the source program is executed immediately after translation.

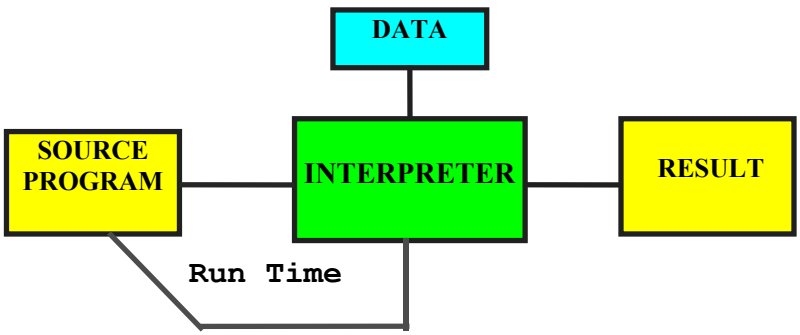

Figure 1-3
The process of translating source code by an interpreter. Examples of interpreted languages are APL, LISP, BASIC and Small Talk. Note that run time and translation time overlap each other.

On the other hand, a **compiler** translates a source program into a particular target language that is usually a computer‰machine or assembly language. The time at which the source program is converted to the object program is called the **compile**

time. When the object program is machine code, it is executed at **run time.** Figure 1-4 illustrates the process of compilation.

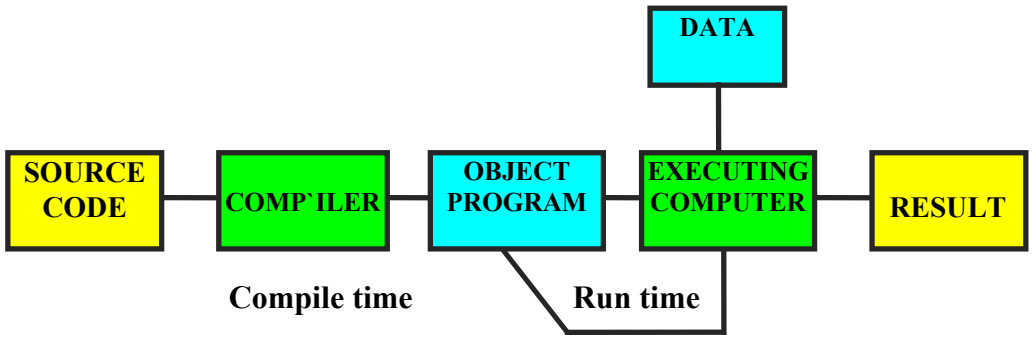

Figure 1-4
A compiler translates the source code in a program all at once to create an executable object program. Translation is only done once. Therefore compilation (translation) time completes before run time begins.

HIGH LEVEL LANGUAGE TRANSLATORS

In order for a Chinese speaking person to be understood by an English speaking person, for example, the Chines language must first be translated to English for it to be understood by the English speaking person. Computers also require this process of translating one language to another. Since machine language is the only language that a computer can understand, all other languages must be translated to machine language. A high-level language translator can be either a **compiler** or an **interpreter.**

An interpreter is a high-level language translator that allows the user to execute a program by translating the high-level instructions one line at a time. A program written in an interpreted language need to be translated each time the program is to be executed. BASIC and FORTRAN are examples of interpreted languages.

HOW DOES A COMPILER WORK?

A compiler is a translator that maps strings into strings. A string is a sequence of characters taken as a single unit. I.e. it converts an input set of string values into another set of string values. The input to the compiler is the source code consisting of a string of symbols. The output from the compiler can be:

- A sequence of absolute machine instructions.
- A sequence of relocatable machine instructions.
- An assembly language program or
- A program in some other target language.

The target language can be that of a supercomputer or another programming language. A relocatable machine instruction is an instruction that reference memory

locations relative to some movable origin. On the other hand, absolute machine instructions have their memory locations fix. Their values can not be changed. The BIOS found on microcomputer is an example of absolute machine instructions.

STAGES DURING COMPILATION

Figure 1-5 illustrates the various stages during the process of compilation of a source program to a target (object) program.

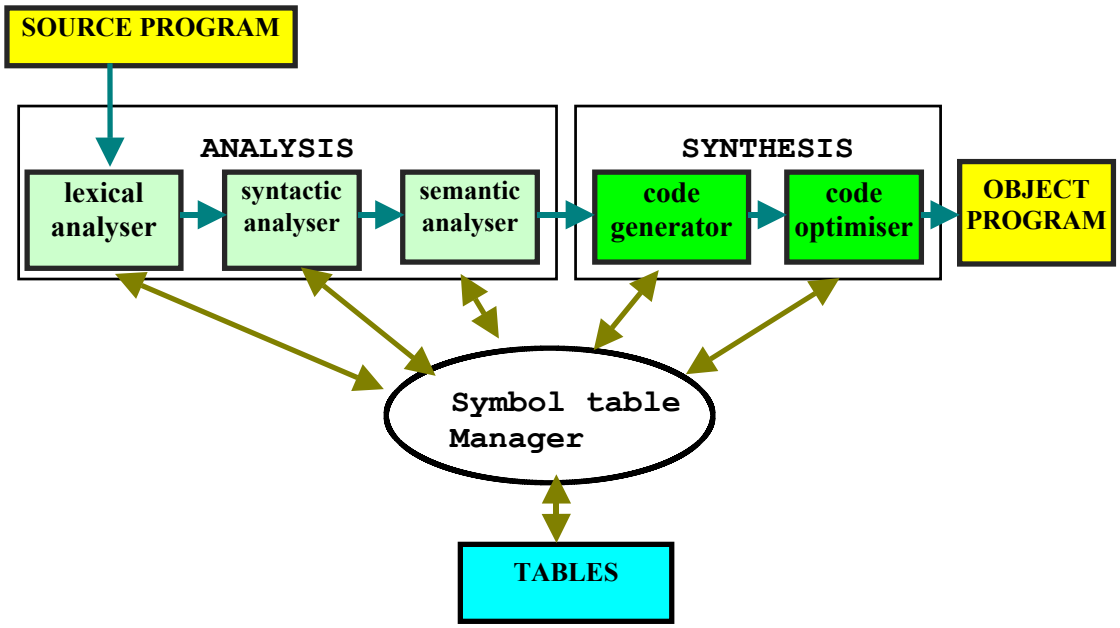

Figure 1-5
Stages during the compilation of a source program to produce an object program.

A compiler analyses a source program and then synthesis its corresponding object program. The source program is decomposed into its basic constituent parts during analysis. The compiler then uses these basic constituents to build the equivalent object program. When the compiler decompose the source code, it creates a table of the constituent basic parts with their characteristic.

A source program is usually a text document consisting of a string of characters. Some characters such as "*****, /, **+**, -, **(**, **)**" are special symbols that represents certain kind of operation. A source program consists of elementary constructs synthesise from a character set used by the language. These elementary constructs can include things such as variable names created by the programmer, constants, and keywords used by the language and operator symbols.

The purpose of the **lexical analyser** is to break down the text that constitute the source program into smaller pieces or **TOKENS** consisting of constants, variable names, operators and keywords. Key words includes commands such as **DO**, **IF**, **WHILE** etc. Syntax analysis is low level analysis. Some analysers (scanner) place

constants, labels and variable names in appropriate tables. For example, a table entry for a variable name may have associated with it, its **Type** (e.g., REAL, INTEGER or BOOLEAN), **object program address, its value** and **the line in the source program where it is declared.** The lexical analyser supplies tokens to the syntax analyser. During syntax analysis, tokens are grouped into larger syntactic classes such as **expressions, statements** and **procedures.** The syntax analyser or parser creates a syntax tree (or its equivalent) in which each leaf represent a token and every non-leaf node represents a syntactic class type.

ANALYSIS OF THE SOURCE CODE

The following briefly summarises the various stages during the analysis of a source program.

1. LEXICAL (or LINEAR) ANALYSIS

During this stage, the text file making up the source program is read from left to right, top to bottom, during which certain characters are grouped together into tokens. Each token has a special meaning and is place in a table.

2. SYNTACTIC (or HIERARCHICAL) ANALYSIS

During this stage, tokens are grouped hierarchically into nested collections.

3. SEMANTICS ANALYSIS

During this stage, checks are made to ensure that the logical structure of the program is meaningful.

The blanks separating each token in an expression are eliminated during lexical analysis. To illustrate **lexical analysis**, consider the following line of code in a high level programming language:

```
Position  := Initial + Rate*60
```

AN EXAMPLE PROGRAM PROBLEM

THE PROGRAM SPECIFICATION

A program is to be designed and then coded that creates an output listing of telephone information on individuals. The input to the program consists of records of data values listed in the table below.

Table 2-10 INPUT DATA VALUES

NAME	TELEPHONE NUMBER	AREA CODE
SAM HORN	749-2138	714
SUE NUNN	663-1271	213
BOB PELE	999-1193	212
ANN SITZ	979-4418	312
END OF FILE	999-9999	999

Figure 2-8
Values in the last record are sentinel. They are used to determine the end of the data file during processing.

The input data file is to be listed as part of the program code for the purpose of testing the program. In this case, an array of record structure is declared in the program and the value in each record is used to initialise the elements of the array structure. The structure of each record is defined with the use of the **struct** statement as illustrated here

```
struct  Tele {int Code; char Num[9]; char Name[12];};
```

See control logic in figure 2-10 and implementation code in figure 2-11. In this declaration, the object identifier; **Code** is a variable name of type **int**, the object identifier **Num** is an array of characters (**char**). This array consists of 9 elements. The object identifier **Name** is also an array of characters with 12 elements. Therefore, the token **Tele** is a structure identifier whose elements are **Code**, **Num** and **Name**. The second **struct** statement at the start of the program coding declares and initialises an array of record type **Tele**. The number of elements in the array is not specified in this declaration. The compiler determines this value at compile time by checking the number of elements available for initialisation of the array. A more detailed and comprehensive explanation on the use of arrays in a C program is given in chapter 4.

The example program illustrated in figure 2-11 creates an output listing of data values stored in a record structure. The record structure is defined within the program code by the use of the **struct** key word followed by a list of member field enclosed in open and closed braces. The **for** statement is used to implement a loop that controls the output listing during the run of the program. The output obtain when the program is run is given below in figure 2-9. The last record in the input data file is used as a trailer record. The trailer record is used by the loop control statement in the **for** instruction to determine the point at which looping is to be terminated. The second

expression in the **for** control statement test for the end of file condition which is used to determine the point at which looping is to be terminated.

OUTPUT FROM THE PROGRAM

The output when the program is run is a listing of the data values in the input record given in the format illustrated below.

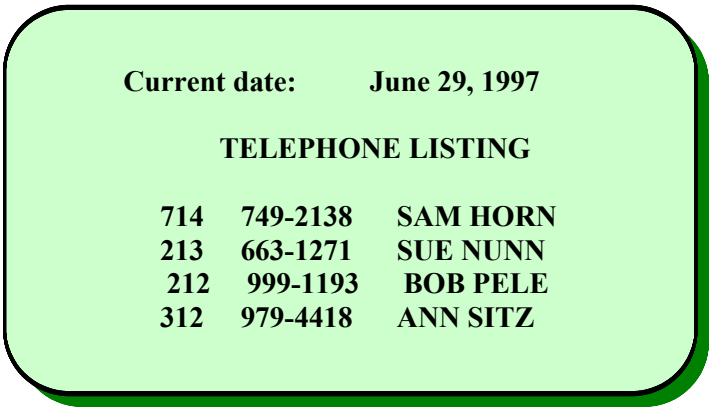

Figure 2-9
Output from the program illustrated by the coding in figure 2-11

THE CONTROL LOGIC

The logic to be used to implement the programming problem solution is illustrated by the flowchart in figure 2-10. Notice that a loop is appropriately used to process the array of records that constitute the input data file. The flowchart is an algorithmic description of the step-by-step process that is to be accomplished in order to implement the solution to the programming problem.

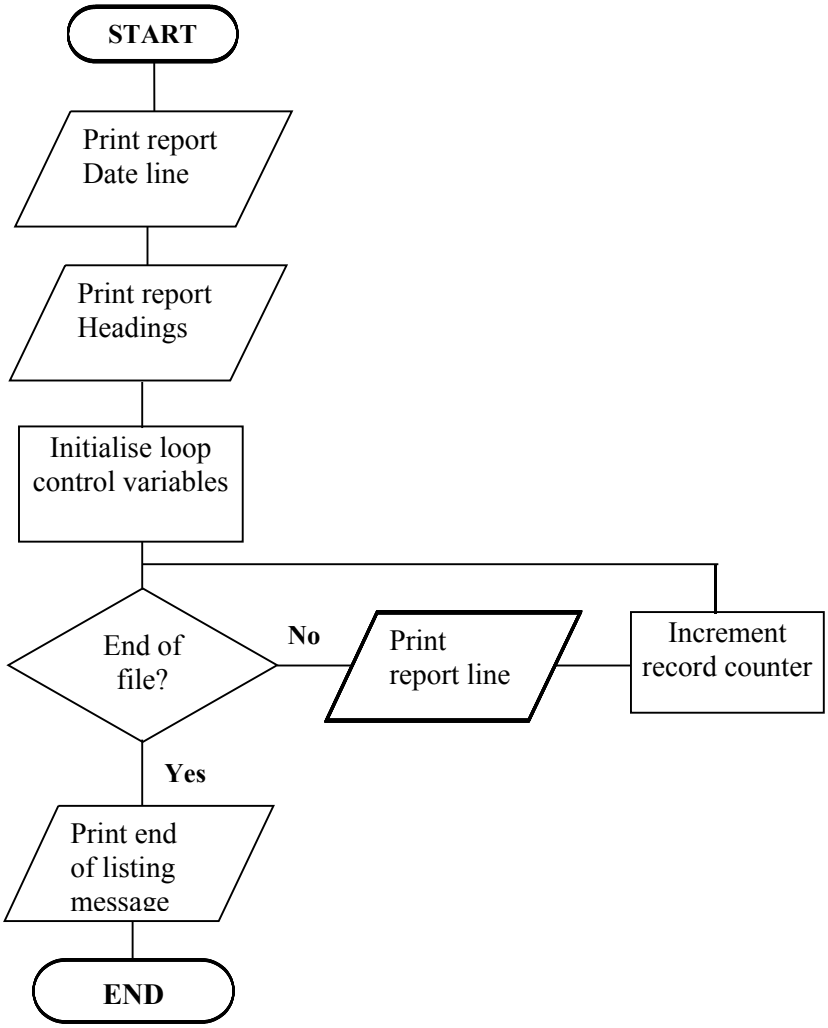

Figure 2-10
The control logic to be used by the source program that implement the given specification. A loop base on a counter variable is used to process the records in the input data file.

THE SOURCE PROGRAM

The C program code that creates the specified output listing is given in figure 2-11. The identifier **TeleFile[ ]** is declared as an array of type **Tele** and is initialised with the listed values in brackets that proceeds it‰declaration. Note that each record in the initialising data values is enclosed within open and close braces. In addition, a comma separates each of these records. The sequence in which values are place in each record matches the data structure definition and no initialising data value is in excess of that allowed by the structure declaration.

```
/*  Input/Output  programming with structure.  D. C. Evelyn,   June 1997     */

#include <stdio.h>
#include <string.h>
#define END    "END OF FILE"

struct  Tele {int Code; char Num[9]; char Name[12];};
struct Tele Telefile[ ] = {{714,"749-2138","SAM HORN"},
                           {213,"663-1271","SUE NUNN"},
                           {212,"999-1193","BOB PELE"},
                           {312,"979-4418","ANN SITZ"},
                           {999,"999-9999","END OF FILE"} };
void main(void)
       {
        int r;

        printf("\n \n \t  Current date:   %20s", __DATE__);
        printf("\n \n \t \t TELEPHONE LISTING \n ");
       for(r = 0; strcmp(Telefile[r].Name, END) != 0;  r++)
          printf("\n\t%i \t%-s \t%s", Telefile[r].Code, Telefile[r].Num, Telefile[r].Name);
        printf("\n \n \t END  OF TELEPHONE LISTING:");
        }
```

Figure 2-11
The program uses the macro __DATE__ to get and print a copy of the DOS date. The program also uses the user defined macro END in the for instruction that test for the end of file condition in the program.

The string function **strcmp( )** is used to compare the string value in the **Name** field of each **TeleFile[ ]** record to the string constant ı **END OF FILE**ˆ. This string constant is defined by the macro **END** as a pre-processor directive at the start of the program. The structure component selector ı **.**ˆ is used to reference individual field in the record structure. It is used in the **printf( )** function in the body of the main loop to reference the field value of each record.

The function **strcmp( )** compares two string values for equality. This function has the following prototype.

```
int strcmp(char *Strg1, char *Strg2);
```

The function returns a non-zero value if the string pointed to by **Strg1** is greater than the string pointed to by **Strg2**. It returns the value 0 when both strings are equal. This prototype is define in the include file ı **string.h**ˆ. Hence this file is included in the source code as a pre-processor directive.

ADDITIONAL PROGRAMMING EXERCISE 2-1

EXC 2-1

1. Add an extra field to the telephone record structure in figure 2-11 so that each telephone record will also show a balance. Declare a **float** variable type with the identifier ı **Balance**ˆ to represent a BALANCE field in each record of the data file.

2. Edit the records used by the program in figure 2-11 so that the BALANCE field for each record has the values listed as follows in each of them. -34.62, 0.02, 621.50, 84.91.

3. You should edit the program code so that the new output from the program when it is run is as illustrated below.

Current date: June 29, 1997

TELEPHONE LISTING

Name		Tele. No.	Balance
SAM HORN	714	749-2138	-34.62
SUE NUNN	213	663-1271	0.02
BOB PELE	212	999-1193	621.50
ANN SITZ	312	979-4418	84.91

END OF TELEPHONE LISTING:

Figure 2-12
New output from the edited program in PRG.EDT2-1.

4. Using appropriate type declaration, use the identifier ı **Total_balance**ˆ to accumulate the balance from each record in the data file during the run of the program. Edit the program logic to show accumulation of all balances in the input data file. The program should then print this value at the end of processing all records in the input data file. Do changes to the program flowchart logic to show this process also.

5. Add the following records to your program data file, then recompile, and run the program.

SORTING THE DATA VALUES

The input values are sorted in ascending order before the operations to determine the averages can be performed. Sorting of the data values in the array can be implemented by any well known sort algorithm such as bubble-sort, bottom up merge-sort, insertion-sort, quick-sort or shuffle-sort. The algorithm for shuffle sort is given in the APENDIX of this manual.

ALGORITHM TO DETERMINE AVERAGE VALUES

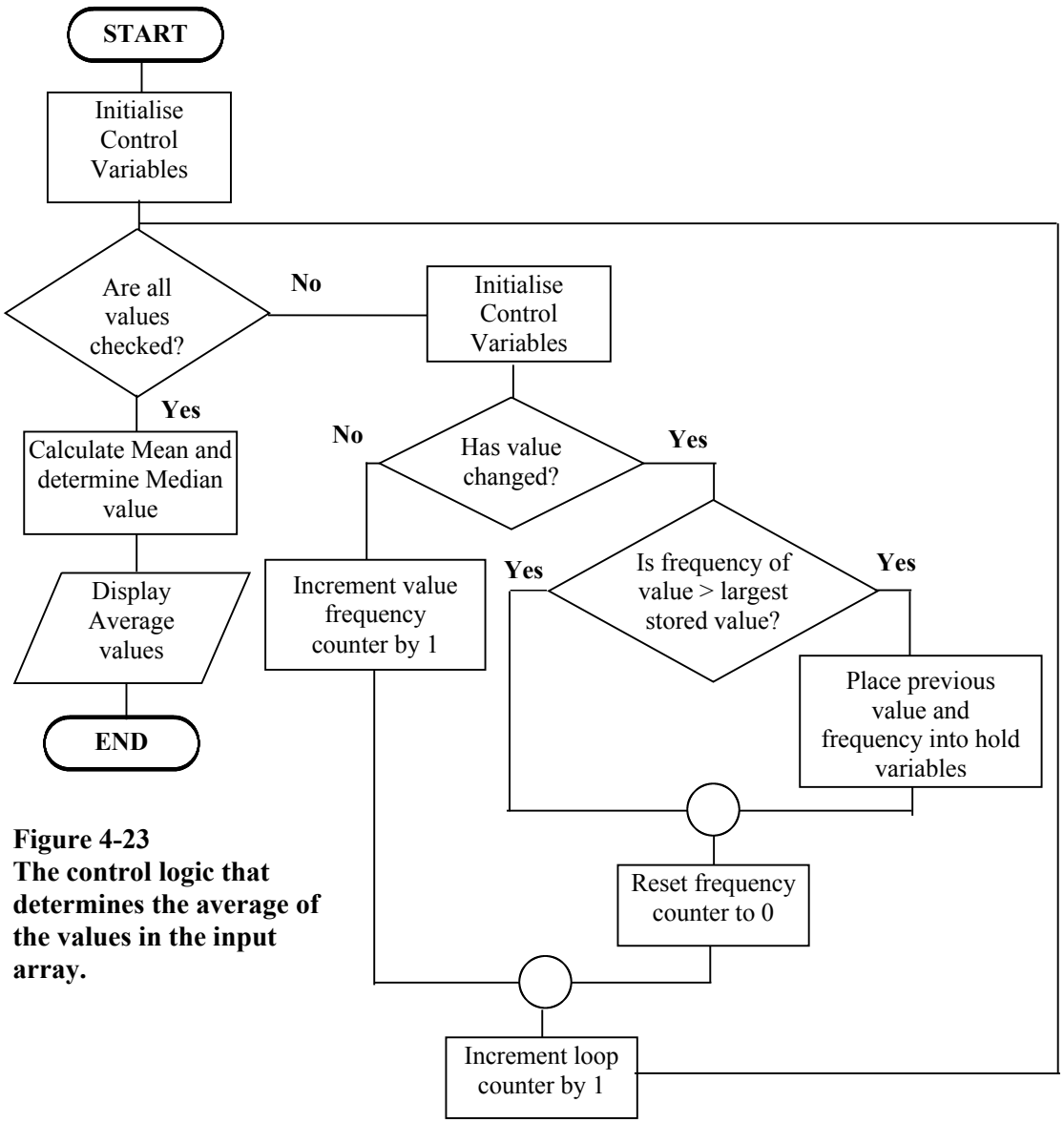

Figure 4-23
The control logic that determines the average of the values in the input array.

(1) Use the logic of the flowcharts in figure 4-22 and in figure 4-23 to code the source program that implement the specification.

(2) The detailed illustrated in the two previous flowcharts does not show how the mean value is actually determined. It does not show accumulation of the values in the input array. Do modification to the logic illustrated in the flowchart to correct this error. Ensure that the correction appear in your source code. Use LogicCoder to redraw the flowchart and generate the new source program.

(3) Write a C program that will determine if a 1-dimensional array is symmetrical. The array is symmetrical if the last value is equal to the first value and the second value is equal to the next to last value and so on. Determine appropriate test data values for the purpose of testing your program. Code the necessary test data into your source code.

(4) Design and then code a simple program that will keep tract of monthly expenditure using a chequebook. The program accepts inputs by taking single character commands followed by a numeric value. The character commands are (**b**) Set starting balance (**d**), deposit (**s**), service charge and (**w**) withdrawal. The program should print the name of the transaction, the amount involved in the transaction and the balance at the end of the transaction. The balance should be calculated at the end of each transaction.

(5) Do analysis of the following and then answer the questions that follows. The two dimensional square matrix **A(n, n) = A** is defined as follows.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

The matrix **A** is symmetrical when for all **i** and **j = 1** to **n**, $\mathbf{a_{ij} = a_{ji}}$. The flowchart in figure 4-24 illustrates the logic of a function that will return the enumerated values FALSE or TRUE when the matrix **A** is not symmetrical or symmetrical respectively. The matrix is represented in memory as a 2-diemnsional array of values.

ALGORITHM TO DETERMINE SYMMETRICAL MATRIX

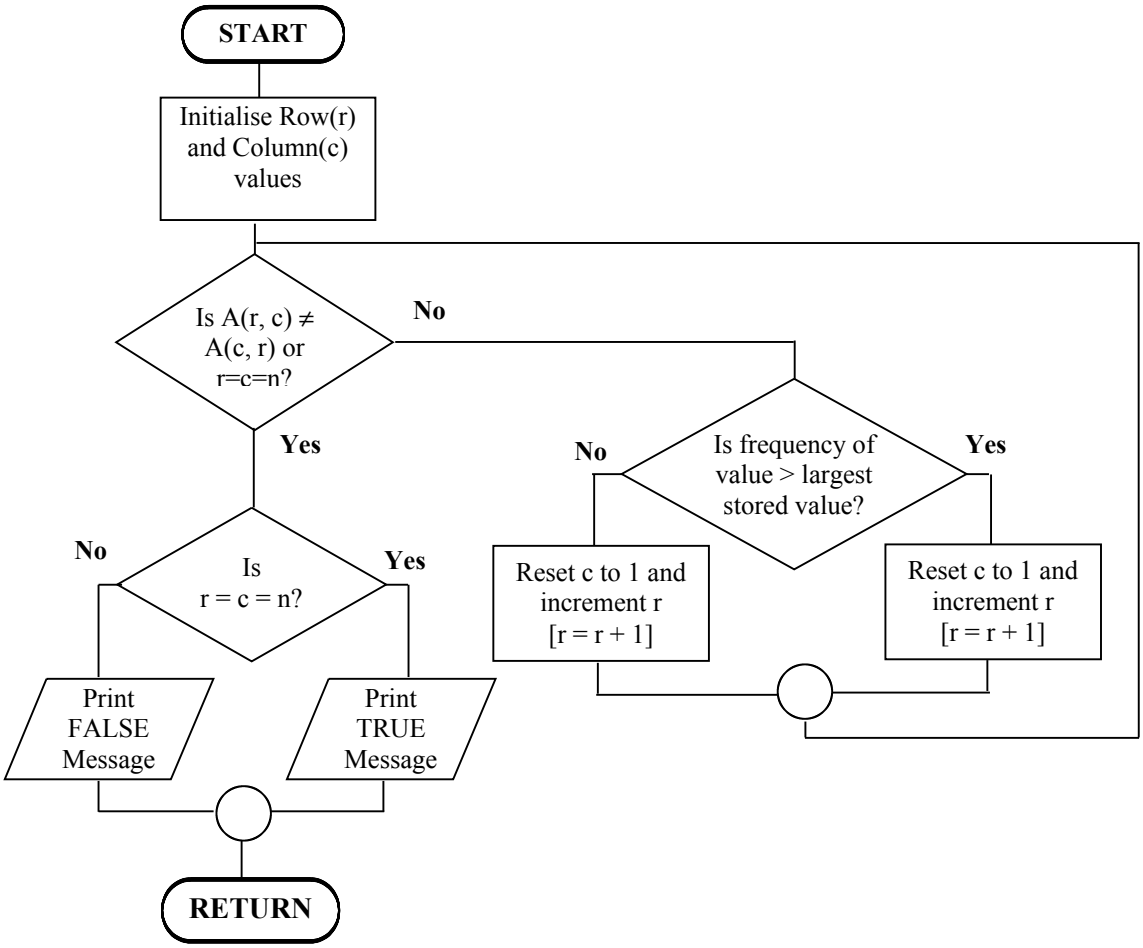

Figure 4-24
Control logic for function that will determine whether or not a 2-dimensional matrix is symmetrical.

Write a C function that will implement the logic illustrated in figure 4-24 above.

ANALYSIS AND QUESTIONAIRE

The logic illustrated by the flowchart in figure 4-24 traverses the matrix row by row so as to identify at least 1 data value $\mathbf{a_{ij}}$ for which $\mathbf{a_{ij} \neq a_{ji}}$ or until all data values in **A** have been checked.

(a) What is the number of test: **A(r, c) = A(c, r)?** when the values
(1) FALSE is returned
(2) TRUE is returned

Give answers for best and worse cases.

HINT: Consider A to be partly consists of the triangular matrix below.

= A is defined as follows.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ & a_{22} & a_{23} & \cdots & a_{2n} \\ & & a_{33} & \cdots & a_{3n} \\ & & & \cdots & \cdot \\ & & & \cdots & a_{nn} \end{bmatrix}$$

You may find the following expression useful in your calculation.
1 + 2 + 3 + . . . k = k(k+1)

(6) Write a function **RemoveCharacter( )** that will accept a pointer to an array of characters, and a character value as input parameters. The function should then remove all occurrence of this character from the character array pointed to. The function should return an integer value that is the number of times the removed character is found in the character array. Blank space should not appear at locations in the string where the selected character has been removed.

(7) Write a function **PalindromeCheck( )** that test a sequence of characters to determine if it is a palindrome. A palindrome is a word that reads the same both forward and backward.

(8) Write the function **ReverseString( )** that will reverse a string of characters without the use of a second array of characters.

(9) Write the function **ChangeCase( )** that will accept a string pointer and an integer value that inform the function to convert all characters in the string to upper or lower case. The function should skip all non-alphabetic characters in the input character array.

THE PROGRAM SOURCE CODE

The following example program creates a list of customers and the outstanding videos they have borrowed from a club. A nested data structure is used to create the record structure for customer records in the database system. Arrays and the assignment operator is used to load the customer records with the input test data used by the