CHAPTER 1

INTRODUCTION

The introduction of high-speed digital computers in the last few decades has provided a very powerful tool for solving a diverse class of problems. The computer, of course, was not the first tool for general purpose problem solving. For centuries people have been developing tools to solve problems. For example, the abacus, the slide rule, and the desk calculator. However, due to the high speed and capacity of modern computers, problems that would have required days, months, or even years to solve can be solved in seconds, minutes, or hours.

In order to use a tool, it is necessary to learn the operating characteristics and conventions of the tool. This is true whether the tool is the abacus, the slide rule, the desk calculator or the computer. It is relatively easy to learn to use an abacus, slide rule or desk calculator. The computer, however, is much more complex and requires the knowledge of a large number of characteristics and conventions.

In order to use a computer as a tool, it is necessary to communicate with the computer   in its operation. This is analogous to depressing the addition, subtraction, multiplication, or division key of a desk calculator. For example, by depressing the addition key, you are instructing the calculator to form and display the sum of the value keyed on the calculator and a current value accumulated by the calculator. The computer also needs this type of instruction. In fact, computers provide all of the arithmetic operations of the desk calculator, as well as providing many more operations.

Instructions are communicated to a computer via an internal language that the computer interprets and reacts to according to its operating characteristics and conventions. This internal language is generally termed a <u>machine language</u>. A <u>program</u> is an ordered collection of instructions that performs the solution of a problem. Again, this is analogous to the sequence and ordering of steps in using a desk calculator to achieve a desired value. The person who writes a program is called a programmer. We shall consider the components and machine language of a simplified computer here in the introduction. However, first let us consider some of the basic characteristics of programming.

1-1 CHARACTERISTICS OF PROGRAMMING

Programming occurs very frequently in our everyday life. That is, we program our activities for certain time periods such as seconds, minutes, hours, days, weeks, months, years, or even for the remainder of our life time. In essence, programming involves developing a plan of action. Frequently, we program our own activities in a very informal manner. Sometimes we take the time to write down the steps of our plan. In most plans, we try to think of contingencies that may arise in the execution of the plan, and in this case, develop alternative plans. The programming of a computer involves the development of a plan. However, this plan, (i.e. the program) must be stated in a formal manner where provisions are made for all contingencies. Before we discuss computer programming, let us consider a program for a process which is well known to all college students. The plan in this case involves preparing for academic course registration.

COURSE REGISTRATION

Assuming that we have the appropriate brochures and registration material, let us consider writing down some steps of a simplified plan for selecting the courses for a semester. As all college students know, many contingencies and frustrations usually arise in the registration process.

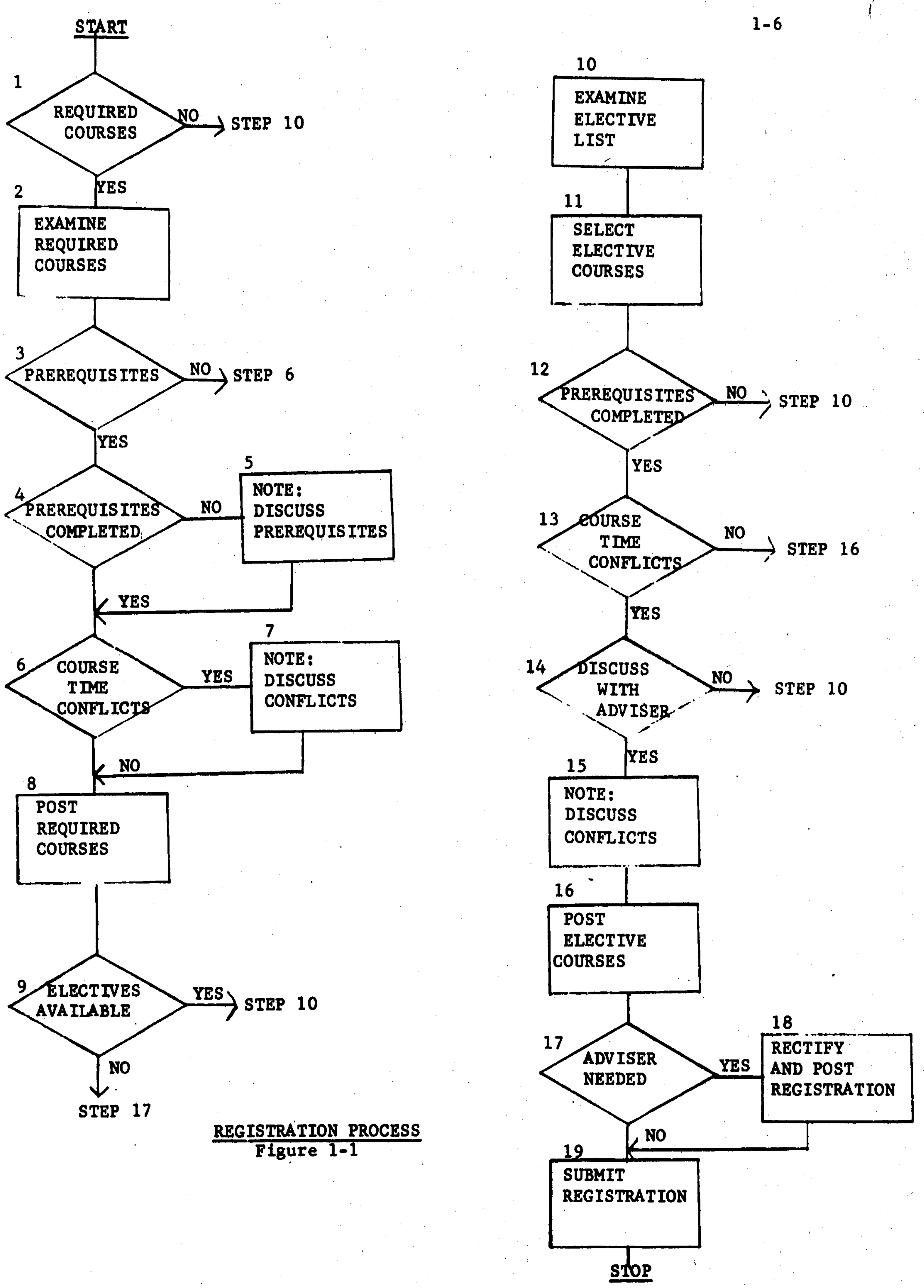
1. Are there any required courses for this semester? If yes continue at step 2, otherwise continue at step 10.
2. Examine the list of required courses.
3. Are there any prerequisites for the required courses? If yes, continue at step 4, otherwise continue at step 6.
4. Have I completed the prerequisites? If yes, continue at step 6, otherwise continue at step 5.
5. Make a note to discuss prerequisites with an adviser.

6. Are there any course time conflicts for the required courses? If yes, continue at step 7, otherwise continue at step 8.
7. Make a note to discuss course time conflicts with an adviser.
8. Post the registration for required courses which can be taken.
9. May I take any electives this semester? If yes, continue at step 10, otherwise continue at step 17.
10. Examine list of elective courses.
11. Select elective courses.
12. Have I completed the prerequisites? If yes, continue at step 13, otherwise return to step 10.
13. Are there any course time conflicts for the elective courses? If yes, continue at step 14, otherwise continue at step 16.
14. Do I wish to consult an adviser about conflicts? If yes, continue at step 15, otherwise return to step 10.
15. Make a note to discuss elective courses with an adviser.
16. Post the registration for elective courses which can be taken.
17. Is an adviser needed? If yes, continue at step 18, otherwise continue at step 19.
18. Rectify scheduling problems with the adviser and post registration for all courses.
19. Submit registration forms to the Registrar's Office.

In this simplified step by step plan, we have considered several contingencies in the registration process. It is by no means exhaustive. The writing of this step by step procedure, is a program. This program is not in the form that would be acceptable for processing by a computer. However, we shall learn that this step by step specification is required at a more detailed level in constructing programs.

As an alternative to writing the above step by step procedure, we can construct an annotated diagram which shows the basic step by step process. This diagram is called a flowchart. In a flowchart, we use certain geometric objects to represent various types of activities of the steps of a program. A flowchart of the registration process appears in Figure 1-1. In this flowchart, a diamond shape is used to denote a decision, and a rectangle is used to show some action to be executed. Each object in the flowchart has been identified with the corresponding step in the written procedure. Where lines are not directly connected, the next step is clearly indicated. You will also notice that the paths leading from all decisions are clearly labeled as "YES" and "NO".

Now that we have considered the basic nature of programming activities, let us consider the characteristics of a simplified computer and a program for the computer.



REGISTRATION PROCESS
Figure 1-1

1-2 BASIC COMPUTER COMPONENTS AND OPERATION

A digital computer is an electronic device composed of several interconnected units. A modern computer system is pictured in Figure 1-2.

The basic components of the computer system may be illustrated in a diagram form in order to understand the interrelationship of the various component units. These interrelationships are portrayed in Figure 1-3. Let us consider the usage of each of the components in the diagram.

There may be one or more input devices which will provide the data to be processed. For example, there may be punched card readers, magnetic tape units, disk units, teletypewriters, etc. In the diagram, a dotted line connects the input unit to the memory unit. Through the execution of an input instruction, data is transmitted from the input device into a portion of the computer memory unit. When the data values are in the computer memory, instructions may be used to calculate the desired results. After the desired results have been obtained, data may be transmitted via output instructions to an output device as illustrated by the dotted line connecting the memory unit to an output device. The output device may be a card punch, printer, magnetic tape unit, drum unit, disk unit, teletypewriter, etc. As with input devices, there may be more then one output device connected to the computer.

In performing the desired calculations with data value in the computer memory, instructions are utiltized which cause the data to be used to be transmitted from the memory unit into the arithmetic unit as denoted by the data

(TO BE SUPPLIED)

Figure 1-2
The IBM System/360 Computer

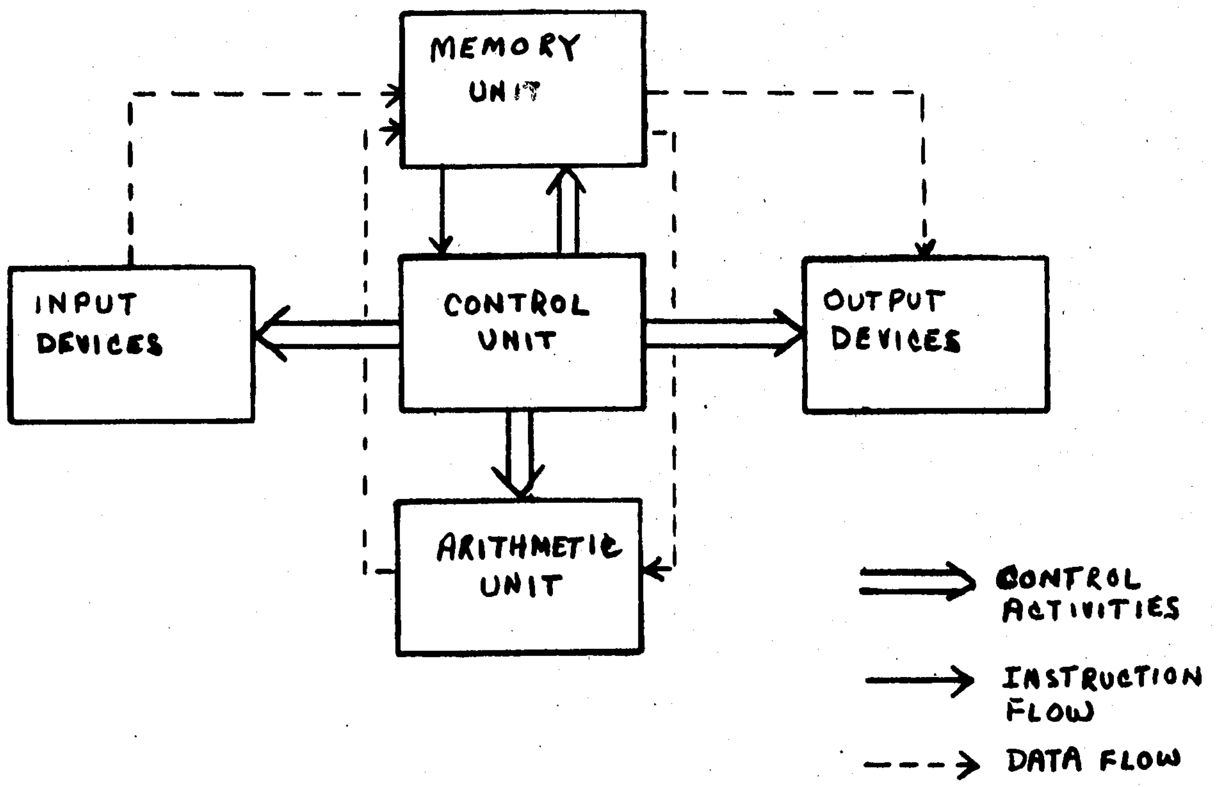


Figure 1-3
Components of a Computer System

flow connection. By the issuance of arithmetic instructions such as add, subtract, multiply, divide, etc. a calculated value is created in the arithmetic unit. The calculated results may then be transmitted back to the computer memory unit ready for further processing or to be placed onto an output device.

At the center of the computer system is a <u>control unit</u>. This unit interprets the instructions of the program and activates the various units in the computer system to execute the instructions of the program. This control activation activity is denoted by the double solid lines.

The instructions of the program as well as the data to be processed are contained in the memory unit. The control unit obtains the program instructions from the memory as denoted by the single/solid line. These are the five basic computer components. Now we shall take a closer look at the components for a computer which we shall call SIMPLE.

<u>THE SIMPLE COMPUTER</u>

A computer memory may be thought of as a series of pigeon holes similar to a series of post office boxes. In the computer memory, each box is called a <u>memory location</u> and it is capable of containing some finite data value. Each box is identified by a number called the <u>memory address</u>. The number of memory locations in the computer varies quite widely. Very large computers have thousands, even millions of memory locations. For illustration purposes let us consider a computer with only 32 memory locations as follows:

(0)	(1)	(2)	(3)
(4)	(5)	(6)	(7)
(8)	(9)	(10)	(11)
(12)	(13)	(14)	(15)
(16)	(17)	(18)	(19)
(20)	(21)	(22)	(23)
(24)	(25)	(26)	(27)
(28)	(29)	(30)	(31)

The numbers in parentheses (0 through 31) are the addresses of each of the 32 memory locations. Let us assume that in the SIMPLE computer, each memory location is capable of containing one program instruction or one data value. Further, we shall assume that each instruction is composed of two parts, that is, an <u>operation code</u> and an <u>operand</u>. The operation code specifies which instruction from the machine's instruction repitoire is to be executed. The operand specifies what memory location is to be used in executing the instruction. We may view the format of an instruction as follows:

CODE	LOCATION

Computers store data and instructions in the computer memory is in a binary coded format. The term <u>binary</u> refers to a base two number system where the values the number system are 0 and 1 as opposed to the familiar decimal base ten number system where the values can range from 0 to 9. The binary number system is described in detail in a later chapter. By grouping binary numbers, it is possible to create numbers of a different base, including decimal numbers. We shall use decimal numbers to demonstrate the computers operation since this is a familiar number system.

The arithmetic unit, contains what is called a <u>accumulator</u> which is used for various arithmetic operations. Also, the accumulator may be used to make decisions concerning the values of various data. We shall assume that the accumulator is the same size as a memory location.

The control unit contains a <u>location counter</u> which is initially set to zero. This location counter is used by the control unit to access the instructions of the program. Let us say that a program is stored in memory location 0 through 12. The instruction in memory location 0 is accessed and executed, then the control unit increments the location counter to 1, accesses the instruction in location 1 and executes that instruction. This process continues (i.e. 2, 3, 4, ... 12). However an instruction may be executed which causes the location counter to be set to identify a memory location other than the next sequential instruction. This is called a transfer of control which causes program to resume execution at the instruction identified by the new contents of the location counter.

Now let us consider a small set of instruction operation codes for our SIMPLE computer. Arbitrary decimal numbers have been assigned. Of course for any given computer, a fixed set of codes are used for particular instructions.

SIMPLE Machine Instruction Repitoire

CODE	OPERATION PERFORMED
00 [address]	Stop the computer. If the computer is restarted address will be placed in the location counter so that the next instruction executed will be in the memory location identified by address.
01 [address]	Place the contents of the memory location identified by address into the accumulator.
02 [address]	Add the contents of the memory location identified by address to the contents of the accumulator, and leave the sum in the accumulator.
03 [address]	Subtracts the contents of the memory location identified by address from the contents of the accumulator, and leaves the difference in the accumulator.
04 [address]	Multiplies the contents of the memory location identified by address by the contents of the accumulator and leaves the produce in the accumulator.
05 [address]	Set the location counter to contain address if the contents of the accumulator is zero. Otherwise continue execution of the instruction in the next memory location.
06 [address]	Set the location counter to contain address if the contents of the accumulator is negative. Otherwise, continue execution of the instruction in the next memory location.

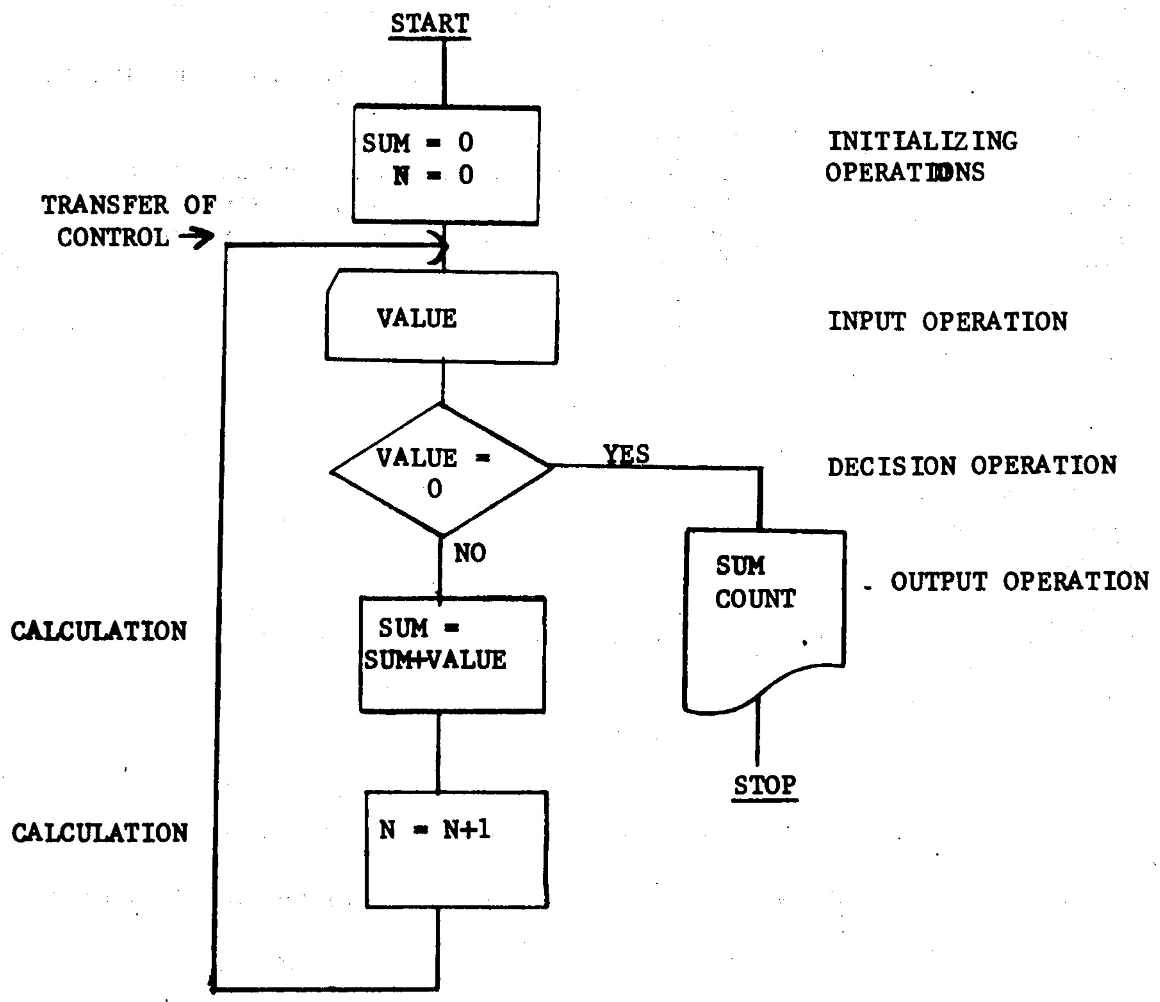
07	[address]	Set the location counter to contain address so that the next instruction to be executed will be the instruction in the memory location identified by address.
08	[address]	Get a data value from an input device and place it in the memory location identified by address.
09	[address]	Put a data value from the memory location identified by address onto an output device.
10	[address]	Store the contents of the accumulator in the memory location identified by address.

Now let us consider a simple program which uses the SIMPLE instructions.

THE PROBLEM

We wish to get a series of values from the input device and calculate the sum of the values, until we encounter a value which is zero. In addition, we wish to count the number of non-zero values processed. When a zero value is encountered we wish to put the sum and the count (N) onto an output device.

As an aid to understanding the logic of this program we shall create a flowchart. The flowchart as introduced in the previous section uses geometric objects to show various types of operations and the order in which the operations are performed. It is generally a good practice to create a flowchart prior to encoding a program. Flowcharting techniques are discussed in detail in a later chapter. The comments in the flowchart specify the type of process performed by each instruction.



The program is contained in memory location 0 through 14. We shall use memory location 29 for the values acquired from the input device; locations 30 and 31 for accumulating the SUM and count (N), respectively. Two locations 15 and 16 are used to contain two constant values, 0 and 1, respectively.

Instead of picturing the instructioa in a pigeon hole memory, we shall simply list the addresses and contents of each of the 32 locations of our SIMPLE computer memory unit.

MEMORY ADDRESS	CODE AND LOCATION	COMMENTS
(0)	0115	Place a zero value in the accumulator.
(1)	1030	Place a zero in SUM location.
(2)	1031	Place a zero in count location (N).
(3)	0829	Get a value and place in location 29.
(4)	0120	Place contents of location 29 in accumulator.
(5)	0512	Resume execution at location 12 if accumulator is zero.
(6)	0230	Add SUM location to the accumulator.
(7)	1030	Store the new SUM.
(8)	0131	Place contents of count (N) in accumulator.
(9)	0216	Add one to the accumulator.
(10)	1031	Store the new count (N).
(11)	0703	Transfer back to get the next input value.
(12)	0930	Put the SUM onto an output device.
(13)	0931	Put the count (N) onto an output device.
(14)	0000	Stop the computer.
(15)	0000	Constant 0.
(16)	0001	Constant 1.
(17)	not used	
⋮	⋮	
(28)	not used	
(29)	Used for Input Values	
(30)	SUM location	
(31)	N location	

To demonstrate the execution of this program, let us consider the following set of input values:

23 -6 0

The following table summarizes the result of executing the program, showing the location-counter, the instruction executed and the contents of memory locations 29, 30 and 31 after the execution of the instruction. An asterisk is used to denote the affect of executing each instruction, where a new value is created in the accumulator or in a memory location.

Instead of using the numeric operation codes, we shall use a three letter code which is representation of the type of operation. This letter representation is called a mnemonic name. Mnenonic means "aiding the memory". For our ten operation codes, the following mnemonics are used.

00	STP	Stop	06	BMI	Branch if accumulator is negative
01	FET	Fetch to accumulator	07	TRA	Transfer
02	ADD	Add to accumulator	08	GTI	Get Input
03	SUB	Subtract from accumulator	09	PTO	Put Output
04	MUL	Multiply by accumulator	10	STO	Store accumulator
05	BZE	Branch if accumulator = 0			

Referenced footnotes point out some significant factors about the program execution. The reader should examine this table closely.

Location Counter	Instruction	Accumulator	Location 29	Location 30	Location 31	Footnote
0	FET 15	0*	?	?	?	(1)
1	STO 30	0	?	0*	?	
2	STO 31	0	?	0	0*	
3	GTI 29	0	23*	0	0	
4	FET 29	23*	23	0	0	
5	BZE 12	23	23	0	0	(2)
6	ADD 30	23*	23	0	0	
7	STO 30	23	23	23*	0	
8	FET 31	0*	23	23	0	
9	ADD 16	1*	23	23	0	
10	STO 31	1	23	23	1*	
11	TRA 03	1	23	23	1	(3)
3	GTI 29	1	-6*	23	1	
4	FET 29	-6*	-6	23	1	
5	BZE 12	-6	-6	23	1	
6	ADD 30	17*	-6	23	1	(4)
7	STO 30	17	-6	17*	1	
8	FET 31	1*	-6	17	1	
9	ADD 16	2*	-6	17	1	
10	STO 31	2	-6	17	2*	
11	TRA 03	2	-6	17	2	
3	GTI 29	2	0*	17	2	
4	FET 29	0*	0	17	2	
5	BZE 12	0	0	17	2	(5)
12	PTO 30	0	0	17	2	
13	PTO 31	0	0	17	2	
14	STP 00	0	0	17	2	

(1) The location counter is initially set to zero, so that is the first instruction to be executed. The activities of the control unit may be

thought of as those a clerk who is handed a labeled set of instructions on cards numbered 0 through 14. He looks at card 0, executes that operation, then card 1, etc. When he finds a card which asks him to make a decision, he will either go to the next card or go to the card indicated in the instruction based upon the outcome of the decision. Also an instruction card may tell him to go to another instruction and resume his work at that point.

This first instruction causes the contents of location 15 to be placed in the accumulator. You will notice that a ? appears in locations 29, 30 and 31. The programmer should make no assumptions about the initial conditions of memory locations. If he wishes to place initial values in particular locations, he must explicitly perform instructions to do this task. Not initializing variables properly is a common programming error. The instructions in locations 0, 1 and 2 are used for this purpose.

When a value is placed into a location via a STO or GTI, it replaces the previous contents of that location. This is referred to as a destructive write operation. The previous contents are lost. When a location is referenced such as FET to accumulator,the contents of the location are not changed, this is called a nondestructive read.

(2) Notice that a decision is made and since the accumulator is not zero. execution continues with the next instruction. In either case, the accumulator and memory location values are not changed.

(3) Control is transferred back to the instruction at location 3, where the next data value is acquired.

(4) Notice that the addition is an algebraic addition (i.e. 23 + -6 = 17).

(5) The zero value has been encountered in the input data so we transfer to the instruction for placing the desired values onto the output device, and stop the program.

Although this program is quite trivial it does demonstrate several of the basic processes in computing. These can be summarized and related back to program instructions as follows:

Basic Process	Instructions
1. Initializing Key Locations	0, 1, 2
2. Value Acquisition (Input)	3
3. Value Calculation	4, 6, 7, 8, 9, 10
4. Decision Making	5
5. Value Disposition (Output)	12, 13
6. Transfer of Control	11
7. Program Termination	14

These processes are inherent in most programs, whether written in machine language, or in a more convenient language such as PL/I, the subject of this book. Through the usage of transfer of control, we can perform iterations in the program. Without this capability, the machine would be almost useless. By using iteration, we are able to utilize the same program segments for processing many data items which is the primary purpose of the digital computer. If we add three more basic processes to this list, we shall have the entire complement of basic programming activities.

8. Value Declaration and Allocation
9. Exception Conditions
10. Program Segmentation

Value declaration and allocation is a means of specifying what data is to be stored in the computer memory and the characteristics of the data. When some conditions arise during the execution of the program (such as creating a value which is too large to be contained in an accumulator), an exception condition is detected and the program must be prepared to handle the condition. The manner in which the program is structured is called program segmentation. The details of how these basic processes are carried out may differ due to various alternative facilities available in the computer and the programming language. Many of the discussions of PL/I programming will refer to these basic processes.

1-3 HISTORICAL COMPUTER DEVELOPMENTS

The computer structure presented in the previous section is an example of a stored program computer. That is, the program as well as the data are contained in the computer memory. The first computer to use a stored program was the EDVAC developed at the University of Pennsylvania in 1949 under the direction of John Mauchly and J. Presper Eckert with the advice of John von Neumann. In fact a stored program computer is frequently classified as a von Neumann type computer. Let us take a brief look at some of the developments in computing prior to and since the development of the EDVAC.

The forerunner of the modern day computer was developed in 1822 by Charles Babbage, a British mathematician. His first invention in this area was a mechanical device called the difference engine which had the capability of adding numbers and displaying results. This device used a series of gears and levers. Then Babbage conceived of an analytical engine which he designed to perform all of the basic arithmetic operations. Unfortunately mechanical skills of that time were not sufficient to develop reasonable components for this type of device. If it had been possible, the device would have undoubtedly weighed several tons. However, many of Babbages ideas were reinvented during the twentieth century in the early developments in electronic computers.

The next major developments in computing came about due to the needs of the U.S. Bureau of Census. In 1890, Hollerith proposed a system involving the usage of punched cards processed by an electromechanical device which could add and sort. From that time up to World War II, these devices were developed by the International Business Machines Corporation and Remington Rand. Several ingenious methods were found for performing more complex operations on these devices, however, their primary usage was in business data processing. In the late 1930's, Stibitz and several others at the Bell Telephone Laboratories developed an electromechanical relay computer to perform complex mathematical computations.

International wars seem to provide a strong stimulus for technological advances. World War II brought about many new developments in the field of electronics, including radar and sonar as well as the advent of the electronic computers. The main electronic technological advance was the development of reliable components. During the war, Howard Aiken at Harvard, in conjunction with IBM developed an extremely sophisticated electromechanical computer called MARK I, which was first demonstrated in 1944. Eckert and Mauchly, taking advantage of the new reliable electronic components, constructed an all electronic computer called the ENIAC in 1946. Then this same group,with von Neumann, developed the EDVAC with the stored program concept. Eckert and Mauchly formed a corporation supported through government contracts and in 1951, delivered the UNIVAC I to the U. S. Bureau of the Census. The UNIVAC I was the first computer to be comercially marketed.

After the war, there were several groups getting into the computer meliu. The extremely fast Whirlwind computer developed at MIT was operational in 1949. Some experimental computers were developed in England during this time period. In 1952, a series of computers started to appear with such interesting names as ORDVAC, ILLIAC, and JOHNIAC.

Most of the postwar computers were using electrostatic tubes as components, until Forrester at MIT developed a magnetic core storage for the Whirlwind. Since then, practically all computers have utilized this type of memory unit. It is forecasted that sometime in the future, the impact of laser technology will affect the manner of storing information in computer memories.

The International Business Machines Corporation, which by far has been the most successful computer manufacturing firm, completed its first computer the 701 in 1953. The 701 was then modified to use magnetic core storage and became the 704 which was the first widely available scientific computer. In addition, IBM developed the 650 in the early 1950's which uses a rotating magnetic drum memory unit. The Eckert-Manchly Corporation was purchased by Remington Rand and went on to develop the UNIVAC II which utilized magnetic core memory. Remington Rand then merged with the Sperry Gyroscope Corporation to create the Sperry Rand Corporation. Following IBM's success with the 650, they developed the Solid State 80 and 90 which also used a rotating drum memory.

Many corporations, domestic and foreign, became eager to get into this dynamic new industry. Many have entered and failed due to the large capital requirements for starting in the industry. Today there are several hundred different computers in the market place, with the rapid pace not showing any signs of dwindling. Computers have become larger, faster and more flexible, and this trend will undoubtedly continue coupled with development of better computer components.

1-4 THE USAGE OF PROGRAMMING LANGUAGES

In the past decade, considerable attention has been directed towards automating the manner of communication with a computer. These efforts have resulted in the creation of programming languages. By utilizing a programming language, it is possible to avoid many of the detailed aspects of utilizing a computer. This enables the programmer to concentrate more on the solution of the problem, rather than the characteristics and conventions of the computer.

Some programming languages reflect computer characteristics more than others. In general, these programming languages that reflect computer characteristics in a direct fashion are termed assembly languages. Programming languages have also been developed that require little or no knowledge of computer characteristics. This type of language is called a procedure-oriented language. All of the languages mentioned have conventions and operating characteristics associated with their usage. They simply represent tools that utilize the computer as a tool. This again is analogous to using the arithmetic functions of the desk calculator to provide a new tool. For example, consider adding a key and set of conventions to the desk calculator that permit the determination of a square root. The calculator would probably use its existing arithmetic capabilities in proper sequence to provide the square root, but the user of the calculator would have a new tool at his disposal.

The term program as previously defined is still appropriate in the context of a programming language. The ordered collection of instructions representing a program that solves the problem are simply stated in the programming language. Again, the person who writes a program may be called a programmer. In the usage of procedure-oriented languages, the programmer may be a physicist, chemist, mathematician, psychologist, soliologist, economist, accountant, etc. This is due to the fact that the procedure-oriented language permits the programmer to specify the solution of his problem in terms with which he is familiar.

Many procedure-oriented languages have been developed in the relatively brief history of computers. Three of the best known languages are FORTRAN, COBOL, and ALGOL. The FORTRAN language (FORMULA TRANSLATOR) was developed primarily for usage in the solution of mathematical problems. COBOL (COMMON BUSINESS ORIENTED LANGUAGE) was developed primarily for the solution of commercial-type problems. These two languages, by far, have been the most heavily used procedure-oriented languages. ALGOL (ALGORITHMIC LANGUAGE) was primarily developed as a convenient language for the natural expression of problems. ALGOL has not been used as widely as the other languages. However, many universities, particularly in Europe, use ALGOL. The three procedure-oriented languages mentioned previously have many common characteristics, as well as providing unique characteristics that are oriented towards the goals of the respective languages. For several years authorities in the field of programming languages have proposed a single all-purpose procedure-oriented language. Representatives of the International Business Machines Corporation cooperated with representatives of the SHARE and GUIDE organizations* and specified a procedure-oriented language called PL/I. The PL/I programming language provides the basic characteristics that are common to a large number of procedure-oriented languages, as well as offering many of the unique characteristics provided by predecessor procedure-oriented languages. PL/I will be used in this textbook as the basis for illustrating problem solving using a programming language.

The programming language and its usage in the framework of a particular computer is the vehicle which is used in problem solving. The user of the programming language should not look for any mystic powers inherent in the programming language or computer to automatically provide problem solution. The engineer, physicist, economist, accountant, etc., must understand the basic nature of the problem he is solving. He must then be able to formulate the steps necessary to compute a solution to the problem. At this point the programming language is used to express the formulation of the problem solution.

The general theme of the book will be the usage of the PL/I programming language as a tool to solve problems from a variety of subject areas. Many examples and exercises have been extracted from textbooks in common usage in many colleges and universities. In utliizing this approach, the student will learn to express the solution of problems (using PL/I as a tool) in terms that reflect the type of problem with which the student is familiar.

The textbook is organized into two parts. Part I introduces the conventions and operating characteristics of the PL/I programming language. We learned earlier in this section that a programming language such as PL/I

*SHARE and GUIDE are groups of customers of the International Business Machines Corporation who make recommendations to IBM concerning their needs as users of IBM equipment.

is simply a tool that utilizes the computer as a tool. The existence of a translator (which is a program) that translates PL/I instructions into machine language instructions, effectively converts a general purpose computer into a PL/I Machine. Consequently, in Part I, we shall, in fact, be learning the conventions and operating characteristics of the PL/I Machine. Many problems will be presented and solved for the PL/I Machine in Part I. In the discussion of the problems, interesting programming techniques will be presented. By examining the problems carefully, the reader shall not only learn how to use the PL/I Machine, he will learn how to use the PL/I Machine effectively. Programming technique is important.

The problems and the exercises of Part I are very general in nature, and in most cases should be understood by all readers whether they can be engineers, mathematicians, students of business, social sciences, or any other academic discipline. In Part II of the textbook, problems from several specialized subject areas will be discussed in depth. The reader should study the chapters of Part II which are applicable to his computing needs. Readers who would like to consider the aspects of assembly language programming prior to considering the PL/I Machine should study the material in Appendix I prior to starting Part I. However, this is not necessary. An assembly language called SAL, for the SIMPLE computer discussed earlier in the introduction is presented in this Appendix. A PL/I program is also presented which simulates the translation and execution of programs written in the SAL assembly language for the SIMPLE computer. In addition, there are several suggested exercises to be programmed in the SAL assembly language. Hopefully, you will have access to a computer center where this simulator can be utilized so that you may gain some basic assembly language programming experience. If not, the encoding of the exercises using the SAL assembly language will be a beneficial experience and add to your appreciation of the future use of a procedure-oriented language.

PART I

CONVENTIONS AND OPERATING CHARACTERISTICS

OF THE PL/I MACHINE

CHAPTER 2

CONCEPTS and DEFINITIONS

In order to demonstrate the structure of a PL/I program, let us consider the simple program we constructed in section 1-2 , as it would be encoded using the PL/I programming language.

```
(1)  SUMMATION:   PROCEDURE OPTIONS (MAIN);
(2)               SUM, N = 0;
(3)       NEXT:   GET LIST (VALUE);
(4)               IF VALUE = 0 THEN GOTO FINAL;
(5)               SUM = SUM + VALUE;
(6)               N = N + 1;
(7)               GOTO NEXT;
(8)      FINAL:   PUT DATA (SUM, N);
(9)               STOP;
(10)              END SUMMATION;
```

The numbers to the left of the program are not part of the program; they are for reference purposes.

The reader should refer to the flowchart of this problem in section 1-1 and compare the machine language program with this PL/I program. The basic instructional unit of a PL/I program is called a <u>statement</u>. Each line (1) through (10) contains a statement / except line (4) which contains two statements. Each statement specifies one or more of the ten basic programming activities mentioned in chapter 1.

The computer does not understand a programming language such as PL/I directly. The computer only understands its machine language. Consequently, the statements of the program must be translated into mahcine language instructions. The manner in which this translation is accomplished will be considered later in this chapter. For now let us consider as the result of translating this program, the equivalences between the PL/I program statements and the SIMPLE machine language instructions discussed in section 1-2.

MEMORY ADDRESS	MNEMONIC CODE	LOCATION	PL/I PROGRAM STATEMENT		
(0)	FET	15	(2)		SUM, N = 0
(1)	STO	30			
(2)	STO	31			
(3)	GTI	29	(3)	NEXT:	GET LIST (VALUE);
(4)	FET	29	(4)		IF VALUE = 0 THEN
(5)	BZE	12			GOTO FINAL;
(6)	ADD	30	(5)		SUM = SUM + VALUE;
(7)	STO	30			
(8)	FET	31	(6)		N = N + 1;
(9)	ADD	16			
(10)	STO	31			
(11)	TRA	03	(7)		GOTO NEXT;
(12)	PTO	30	(8)	FINAL:	PUT LIST (SUM, N);
(13)	PTO	31			
(14)	STP	00	(9)		STOP;
(15)	0000	} constants created by the translator			
(16)	0001				
(17)	not used				
⋮	⋮				
(28)	not used				
(29)	VALUE	} The translator relates these names to particular memory addresses			
(30)	SUM				
(31)	N				

Statements (1) and (10) are not included in the program. These statements, the PROCEDURE and END statements are used for the purpose of segmenting the program (i.e. specifying the beginning and ending).

The translator related three names (i.e. VALUE, SUM and N) to three memory address (i.e. 29, 30 and 31). These names in programming language lingo are called identifiers. In this case, the identifiers name what are called variables. A variable is a data item in the computer memory whose contents may change during the execution of the program. Identifiers are also used to identify parts of the program which are called statement-labels as in the case of NEXT and FINAL. In this translation, the statement-label NEXT is associated with the memory address 3, and FINAL is associated with the memory address 12. The name that appears prior to the PROCEDURE statement and following the END statement is called a procedure-identifier. This is the name of the program. For this program, we have chosen the identifier SUMMATION.

In lines 2, 4 and 6 of the PL/I program the values 0 and 1 were utilized. These values in programming language lingo are called constants. That is, their value will not change during the execution of the program. The translator has placed these constant values in memory locations 15 and 16.

The individual statements of the program have in many cases been translated into more than one machine language instruction. This is quite typical. As you examine the PL/I program in relationship to the machine language program you will see why a procedure-oriented language is easier to utilize. It frees the programmer from the burden of understanding the details of the machine language. The particular PL/I translator used assumes the burden of knowing the machine language. The reader should trace through the PL/I program to be satisfied that these two programs perform the same activities. In performing this trace, consider the following basic programming activities which are performed by the various PL/I statements.

STATEMENT	PL/I STATEMENT TYPE
(1) Program Segmentation	PROCEDURE
(2) Initialization of Variables	assignment*
(3) Value Acquisition	GET
(4) Decision Making	IF
(5) Value Calculation	assignment*
(6) Value Calculation	assignment*
(7) Transfer of Control	GOTO
(8) Value Disposition	PUT
(9) Program Termination	STOP
(10) Program Segmentation	END

*The assignment statement specifies that the contents of the variables to the left of the = sign are to be replaced by the value created on the right of the = sign in a destructive write operation.

For example:

SUM, N = 0;	Places a zero value in SUM and N.
SUM = SUM + VALUE;	Adds the contents of SUM to the contents of VALUE and stores the result in SUM. The contents of VALUE are unchanged after the statement is executed. The previous contents of SUM are destroyed.

The portion of the assignment statement to the right of the equal sign is called an expression. Basicly, an expression is a formula which specifies how a value is created as in SUM + VALUE. Expressions are extremely important in the programming language and they will be discussed in detail in the next chapter.

The various types of statements in the PL/I program with the exception of the assignment statement are identified by various names (i.e. PROCEDURE, GET, IF, GOTO, PUT, STOP and END). These names in programming language lingo are called statement-identifiers. As you undoubtedly have observed, the statement-identifiers are closely associated with the basic programming activity they perform. Now let us consider some of the factors in preparing a PL/I program for translation and execution.

2-1 SOME BASIC SYNTAX ELEMENTS

After the problem has been analyzed, and a flowchart has been created, the program must be written in terms of the programming language. As with natural languages, such as English, French, German, Spanish, etc., the programming language has established rules and conventions concerning the usage of the language. The rules of a programming language or a natural language are referred to as the syntax rules of the language. The basic components of a natural language are the symbols used in the formulation of words and meanings. Further, in most natural languages there are rules concerning the construction of groups of words to form a meaningful sentence or expression. The programming language also specifies rules equivalent to the rules for symbols, words and sentences. However, in the programming language, the terms characters, identifiers and statements are utilized.

CHARACTERS

The symbols used in constructing a program are called characters.

The PL/I programming language character set contains 60 graphic characters. These symbols form the basis for constructing programs using the programming language. The special purposes for which the characters are utilized in the programming languages are introduced as required throughout the book. The 60 characters are as follows:

The English upper-case letters

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z

The decimal digits

0,1,2,3,4,5,6,7,8,9

The following additional characters

=	Equal symbol	¬	Not symbol
+	Plus symbol	&	Ampersand
-	Minus symbol	|	Vertical stroke
*	Asterisk	>	Greater Than symbol
/	Slash	<	Less Than symbol
(	Left Parenthesis	_	Break Character
)	Right Parenthesis	?	Question Mark
,	Comma	□	Blank (no graphic)
.	Period or Decimal Point	$	Dollar sign
'	Quotation Mark	@	Commercial At symbol
%	Per Cent symbol	#	Number symbol
;	Semicolon		
:	Colon		

The character blank which does not have a graphic character will be denoted throughout the book by a rectangular box □ . The graphic symbols used in the programming language must be conveyed to the computer in a coded form. The most common encoded form of graphic characters is the familiar punched card which was invented by Hollerith as discussed in the previous chapter. The punched card in Figure 2-1 illustrates the encoding of the 60 characters that may be used in the programming language. This card is prepared using a standard IBM 029 key punch machine. Notice that there are 80 columns on the card, each of which may be used to represent a single character. The holes punched in each column represent the encoding for the characters displayed at the top of the column with the exception of the blank character in column 57.

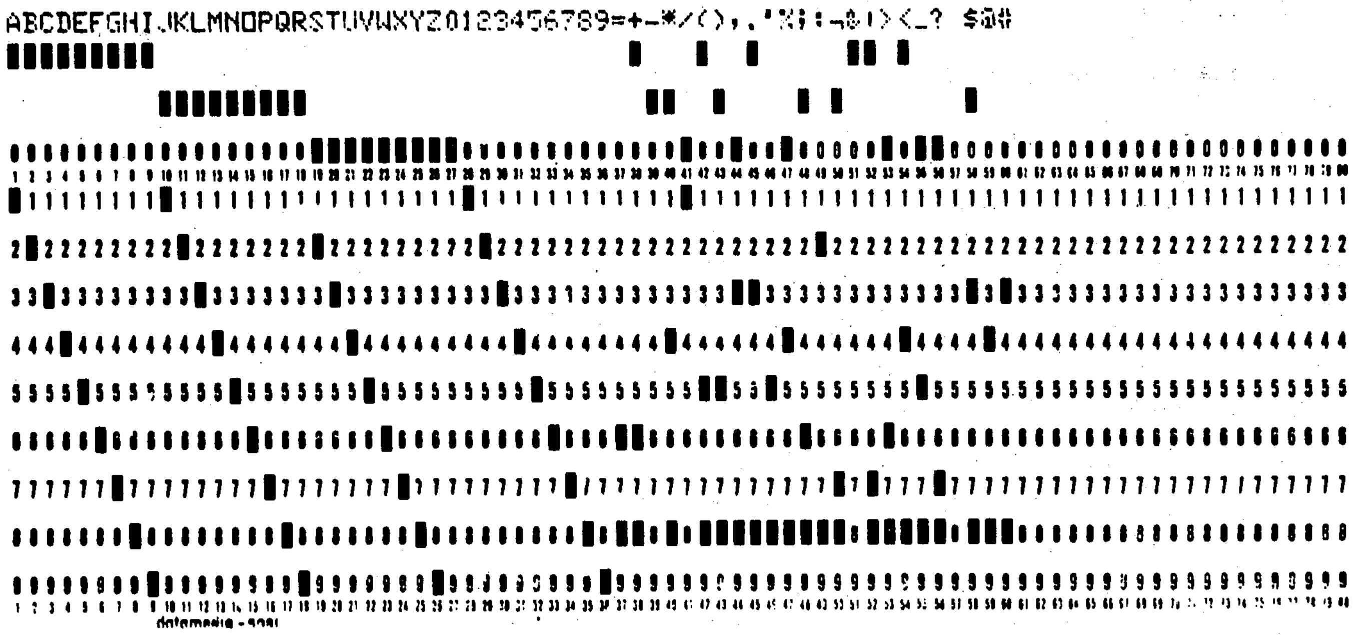

ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789=+-*/(),.'%;:¬&|><_? $@#

Figure 2-1. Punched Card of the Programming Language Characters.

NAMES

Now let us consider the syntax rules used in naming objects in a program. The concept of a name is certainly well-known. Names are used to identify people, actions, objects, concepts, values, etc. For example, when the name JOE is introduced, one thinks of a person, and probably of a particular person with that name. The name CAPITALISM is used to identify a type of economic system, and the name BALL is used to identify a round object. Frequently, it is desirable to use a special symbology rather than a name. This is particularly true in science and engineering. Symbols, like names, may be used to identify actions, objects, concepts, values, etc. For example, consider the following symbology and their associated meaning:

Σ	summation symbol identifying an action
$\bar{x}$	mean symbol identifying an average value
π	pi symbol identifying a constant value (3.14)
H_2O	chemical symbol identifying water
$\sqrt{\ }$	square root symbol identifying an action

In addition to names and symbols that are used for identification, people frequently invent names to identify actions, objects, concepts, values, etc. These invented names may be abbreviations of full names, initial letters taken from several words or any other invented construction that conveys a meaning. This type of name is referred to as a mnemonic name. In chapter 1, we used mnemonic names to identify the basic operations of the SIMPLE computer. Some examples of mnemonic names are as follows:

QE	Quadratic Equation
DET	Determinant
SQRT	Square Root
COC	Coefficient of Correlation
GMT	Greenwich Mean Time
FICA	Federal Insurance Compensation Act
IQ	Intelligence Quotient
AGCT	Army General Classification Tests
TAT	Thematic Apperception Test
SES	Socio-Economic Status
UCS	Unconditioned Stimulus
AFL	American Federation of Labor
MTM	Method Time Measurement
LIFO	Last In, First Out

IDENTIFIERS

In the programming language, names are used for several purposes, including the identification of variables in the computer memory, parts of a program, input and output devices, exception conditions and various other program components. However, as was discussed previously, the term identifier is used in the programming language instead of name. The programming language tool is utilized primarily to specify the manipulation of the values (via their identifiers) in order to produce a desired result. In the SUMMATION program, the identifiers SUM, N and VALUE are used to identify values to be manipulated by the program.

The construction of an identifier must conform to the conventions of the programming language. Specifically, the following rules must be followed:

1. The identifier cannot contain more than 31 characters.

2. The first character used in the identifier must be a letter, i.e., A,B,C......Z, or $,#, or @.
3. The remainder of the identifier may be composed of any letter, any decimal digit, i.e. 0,1,2...9, the underscore character __, $, #, or @.

From the discussions earlier in the chapter, it should be obvious that most of the names and mnemonic names can be written following the identifier construction rules. However, the symbols illustrated, such as $\sum$, $\bar{x}$, π, etc., may not be used directly in the programming language. Identifiers may be constructed to represent these symbols, such as SUM, MEAN, PI, etc. The programmer may use any identifier construction desired, however, he should attempt to have the identifier convey a meaning consistent with the nature of the object being identified. Some representative identifiers chosen from various subject areas are as follows:

```
MENTAL_AGE                     DETERMINANT
CHRONOLOGICAL_COEFFICIENT      VELOCITY
STANDARD_DEVIATION             HEAT_FACTOR
CORRELATION_COEFFICIENT        DISTANCE
POPULATION                     OHM
PERCEPTUAL_SPEED               VECTOR
RORSCHACH_SCORE                SLOPE_PROJECTION
HETEROGENEITY_INDEX
DEBIT
CREDIT
BALANCE
GROSS_NATIONAL_PRODUCT
MARGINAL_RATE
INTEREST_RATE
INVENTORY_ON_HAND
YEARLY_PROFIT
LABOR_UTILIZATION
TURNOVER_RATE
```

IDENTIFIER-VALUE RELATIONSHIPS

The association of a value with a variable is specified in the program. Values normally reflect some unit of measurement. For example, dollars and cents, percentages, velocity, resistance, radians, degrees, number of consumers, number of products, inches, yards, etc. To illustrate the identifier and value of relationships, consider the following:

CHRONOLOGICAL_AGE	6 measurement of years
IQ	110 measurement in percentage
POPULATION	1506000 measurement in number of people
COLOR_SPECTRUM_MAX	700 measurement in millimicrons
SUMMATION_TONE	5500 measurement in frequency (cycles per second)
MOBILITY_INDEX	20.2 measurement in percentage
SIN	60 measurement in degrees
DETERMINANT	.7 density factor of a matrix related to 1.0
DISTANCE	.09 measurement in fraction of an inch
PROTON_MASS	1.00759 measurements in amu's (atomic mass units)
DEBIT	532.75 measurement in dollars and cents
INTEREST_RATE	6.5 measurement in percentage
NUMBER_CONSUMERS	1564 measurement in number of consumers
WORKING_CAPITAL	91965.35 measurement in dollars and cents

The computer does not associate particular units of measurement such as dollars and cents, velocity, etc. with particular variables. The computer simply performs operations involving values. The association of a particular unit of measurement is only known by the programmer and is dependent upon the nature of the program. In the SUMMATION program, we did not specify the units of measurement that were being summed. The values acquired as input could represent population, salaries, amperes or any other unit of measurement. The computer execution of the program is independent of the units of measurement represented by the values.

STATEMENT-LABELS and PROCEDURE-IDENTIFIERS

When identifiers are used as statement-labels, the identifiers used should be suggestive of the nature of the processing performed in by the particular part of the program. In the SUMMATION program, the statement-labels NEXT and FINAL were used. NEXT is used to suggest that that part of the program acquires the next value. FINAL is used to suggest that this part of the program is executed last, after all of the values had been acquired. Statement labels of the form:

```
X1   #2
L5   $ABC
A3   XYZ
```

are permissible statement-labels since they conform to the identifier construction rules, however, they are not very useful in identifying that nature of the processing performed in a part of a program. Consider the following more representative statement-labels.

```
TABULATE               COMPUTE_FICA
INTEREST_FACTOR        TIME_CALCULATION
EVALUATE_QUADRATIC     INTEGRATE
CORRELATE              AVERAGE_CALCULATION
```

The programmer should also select a program name (i.e. procedure-identifier) which reflects the nature of the entire program as in the case of the program SUMMATION. Some representative procedure-identifiers are as follows:

```
ORBITAL_CALCULATION    NONLINEAR_REGRESSION
PAYROLL_PROCESSING     INTELLIGENCE_TESTS
VISUAL_PERCEPTION      POLYNOMIAL_EVALUATION
```

We have now considered the usage of identifiers for three purposes, that is, for variables, for statement-labels and for procedure-identifiers. Other uses of identifiers will be considered later. The programmer can create any identifiers he wishes for these purposes (subject to the identifier construction rules), however, within a procedure delimited by the PROCEDURE and END statements, he may not use the same identifier to identify more than one object. For example, you may not have a variable called SUM and a statement-label called SUM.

CONSTANTS

In the SUMMATION program, we used two constants, 0 and 1. There are several types of constants permitted in PL/I. For now we shall consider only two types, fixed-point decimal constants and floating-point decimal constants.

A fixed-point decimal constant may be an integer or non-integer value composed of decimal digits (0 through 9). If a non-integer constant is to be specified a single decimal point indicated by a period (.) may be placed in the constant.

EXAMPLES:

$$\left.\begin{array}{r}0\\5\\360\end{array}\right\}\text{ integer constants}$$

$$\left.\begin{array}{r}3.14159\\.05\\250.50\\.005982\\3798.5\end{array}\right\}\text{ non-integer constants}$$

The term fixed-point means that the position of the decimal point is fixed, that is, it is either explicitly stated for a non-integer value or it is assumed to appear immediately to the right of the last digit of an integer value. This is not necessarily the case with floating-point decimal constants. Before considering the syntax of floating-point decimal constants, let us consider the nature of floating-point numbers.

FLOATING-POINT NUMBERS

A floating-point decimal number is composed of two parts. Firstly, the actual numeric value of the number which is of fixed length called the mantissa, and secondly a number representing an exponent, that is, the power of 10 to which the value expressed in the mantissa is to be raised. For example, the following is a representation of a floating-point number as expressed in the programming language.

1E2

Where:	1	is the mantissa
	2	is the exponent
	E	separates the mantissa from the exponent

The actual value of this floating-point number can be developed as follows:

1E2

$= 1 \times 10^2$

$= 1 \times 100 = 100$

It should be obvious from this example that the exponent simply shows the positioning of the decimal point. In this example, the decimal point is two places to the right of the mantissa value. Now let us consider a few more floating-point numbers

5E1 =	50 =	5×10^1
150E0 =	150 =	150×10^0
532E3 =	532000 =	532×10^3

where $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1000$, etc.

In the programming language, an actual decimal point may be placed in the mantissa, in which case the decimal point specified by the exponent is relative to this position. For example, consider the following:

2.3E3 =	2300 =	2.3×10^3
.5E2 =	50 =	$.5 \times 10^2$
622.E1 =	6220 =	622×10^1

Thus far, we have only considered a positive exponent which specifies the positioning to the right. It is also possible to specify a negative exponent, in which case, the positioning is to the left. For example, consider the following floating-point values.

1E-2 =	.01 =	1×10^{-2}
.5E-3 =	.0005 =	$.5 \times 10^{-3}$
6.5E-1 =	.65 =	6.5×10^{-1}
532.E-4 =	.0532 =	532×10^{-4}

where: $10^{-1} = \frac{1}{10} = .1 \quad 10^{-2} = \frac{1}{100} = .01, \ 10^{-3} = \frac{1}{1000} = .001$, etc.

Thus you can see, by simply changing the exponent, the range of the value can vary quite widely.

A floating-point decimal constant is composed of one or more decimal digits (0 through 9) representing the mantissa (with an optional single decimal point specified by a period (.)) immediately followed by the letter E. The E is followed by a + or - sign followed by a decimal exponent or simply followed by a decimal exponent in which case, it is assumed to be a positive exponent.

If a decimal point does not appear in the mantissa, it is assumed to be to the right of the mantissa value.

```
EXAMPLES:  314159E-5
           .3141592653589793E1
           2.7182818E0
           12.5E6
           .5963E-18
```

Obviously fixed-point numbers are easier to understand and recognize and in most cases, you will probably select to use fixed-point numbers. Some computers like the IBM System/360 are capable of containing the representation of fixed-point decimal values in the computer memory and performing operations directly upon this type of value. However, in general, it takes more computer time to perform operations on fixed-point decimal values, whereas floating-point values can be manipulated much faster in the computer. Further implications of various types of numbers and additional PL/I constant formats will be discussed the last section and in the next chapter.

STATEMENTS

The next set of syntax rules are used for the construction of statements, the basic instructions of the program.

The general format of a statement is as follows:

[statement-label: statement-identifier statement-body;]

1. One or more statement-labels may optionally appear prior to the statement. Each statement-label that is specified, must be followed by a colon (:).
2. The statement-identifier specifies the instruction or action to be performed.
3. The statement-body contains information required for the execution of the action specified in the statement-identifier. The nature of the information is dependent upon statement-identifier that is specified.
4. The assignment statement and the null statement discussed in the next chapter, are the only statements in the programming language that are specified without a statement-identifier.
5. All statements are terminated by a semi-colon (;).

In the SUMMATION program, two statements were preceeded by the statement-labels, namely NEXT and FINAL. The identifier SUMMATION, although it appears prior to a statement, namely PROCEDURE is a procedure-identifier rather than a statement-label. An instance where it might be useful to utilize more than one statement-label is when a particular segment of the program performs the operations for several cases. For example, consider the following skeleton

```
CASEA:  CASEB:   A = A + B;
       :
CASEC:  A = A - B;
       :
CASED:  CASEE:  CASEF:  GET LIST (A, B);
       :
```

Transfers of control such as:

```
GOTO  CASEA;
GOTO  CASEB;
```

Transfer control to the same program statement.

There are a multitude of statement types that will be considered in this textbook, all of which provide various basic program activities. The following list summarizes all of the statement types to be considered by the type of program activity they perform. You will note that some statement types are listed in more than one category, since they perform more than one basic activity. You shall also notice that some statement-identifiers may be abbreviated. That is, the translator will accept either form. In addition, the GOTO statement may be specified without a blank character between GO and TO or with a blank character (i.e. GO▯TO).

1. Program Segmentation

   PROCEDURE or PROC
   BEGIN
   DO
   END
   ENTRY

2. Value Declaration and Allocation

   DECLARE or DCL

3. Initializing Key Variables

   assignment statement
   DECLARE or DCL

4. Value Acquisition (Input)

   GET
   OPEN
   CLOSE
   READ
   FORMAT

5. Value Calculation

   assignment statement
   PUT

6. Decision Making

   IF
   null statement
   DO

7. Value Disposition (Output)

   PUT
   OPEN
   CLOSE
   WRITE

   FORMAT

8. Iteration and Transfer of Control

   GOTO or GO□TO
   CALL
   RETURN
   DO
   END

9. Program Termination

   STOP
   RETURN
   END

10. Exception Conditions

    ON
    SIGNAL

All of these basic programming activities and several of the associated statement types will be discussed in the next chapter. The other statements will be presented later in the textbook.

KEYWORDS

In the SUMMATION program, you will notice that various words appeared in several of the statement-bodies such as OPTIONS, MAIN, LIST, THEN and DATA. These words are termed keywords. That is, they provide information to the translator as to the structure of statements and the various alternative activities to be performed by various statements. All of the statement-identifiers are also keywords. Many keywords will be introduced throughout the textbook.

The programming language PL/I permits the programmer to use keywords as identifiers for procedures, statement-labels, variables, etc. However, using keywords as identifiers is not recommended, since it reduces the readability of the program. To illustrate this/ point, consider the following version of the SUMMATION program which performs the same program activities but is more difficult to understand.

```
SUMMATION:  PROC OPTIONS (MAIN);
            DATA, N = 0;
      GET:  GET LIST (LIST);
            IF LIST = 0  THEN GOTO PUT;
            DATA = DATA + LIST;
            N = N + 1;
            GOTO GET;
      PUT:  PUT DATA (DATA, N);
            STOP;
            END SUMMATION;
```

The keywords GET and PUT have been utilized as statement-labels and the keywords DATA and LIST have been used as variable identifiers. Do you agree that it is less readable than the original version of SUMMATION? The translator is not confused by this type of multiple usage of words since the context in which the words are used in the program statements determines their meaning (i.e. variable, statement-label, keyword, etc.).

2-2 PROGRAM PREPARATION, TRANSLATION and EXECUTION

After the program has been written, it must be encoded on some medium which is an acceptable input to the computer. We shall consider this input medium to be punched cards which are acceptable to the card reader input device. Other medium such as typewriters which are directly connected to the computer may be used in some systems.

In addition to punching the program into punched cards, we must also prepare cards that contain the data to be processed by the program and what are called control cards. The syntactical format and information content of control cards depend upon the conventions of an operating system. An operating system is a sophisticated machine language program which schedules the work of the computer. We shall assume in this textbook that the operating system OS/360 (1) for the System/360 is to be utilized. The reader should check the local control card conventions used by the computer center at which he intends to have his PL/I programs processed. Since control card conventions vary quite widely we shall not discuss them in detail in this textbook. For a detailed discussion of the control cards of OS/360 and the appropriate conventions for PL/I programs, the reader is referred to (2) and (3). Another document (4) is a reference manual which describes all of the PL/I features that are implemented by the OS/360 PL/I translator.

1. IBM System/360 Operating System
   Concepts and Facilities Form C28-6535.

2. IBM System/360 Operating System
   Job Control Language Form C28-6539.

3. IBM System/360 Operating System PL/I (F)
   Programmer's Guide Form C28-6594.

4. IBM System/360 PL/I Reference Manual
   Form C28-8201.

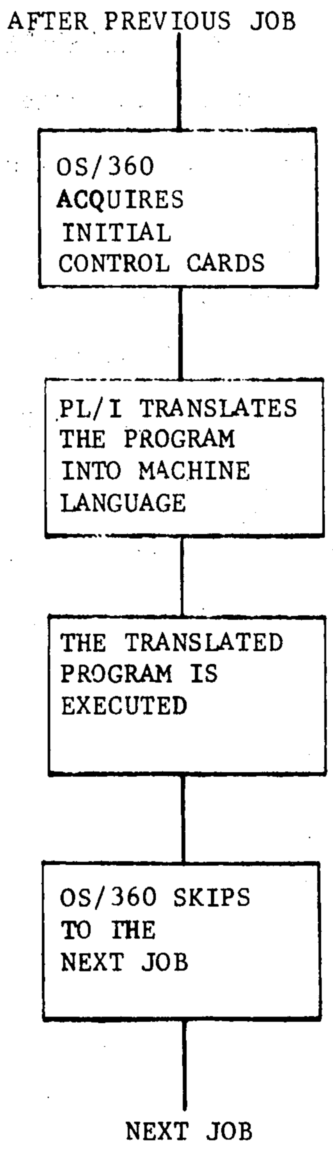
The OS/360 control cards, the SUMMATION program and the input data are prepared on punched cards in the following order.

```
//LF127700 'EE397200', LAWSON
//A EXEC PL1FCLG, PARM.PL1 = 'X,ATR,EB',PARM.LKED = 'LET'
//PL1.SYSIN DD *
  SUMMATION:  PROCEDURE OPTIONS (MAIN);
              SUM, N = 0;
       NEXT:  GET LIST (VALUE);
              IF VALUE = 0 THEN GOTO FINAL;
              SUM = SUM + VALUE;
              N = N + 1;
              GOTO NEXT;
      FINAL:  PUT DATA (SUM, N);
              STOP;
              END SUMMATION;
/*
//GO.SYSIN DD *
  23 -6 0
/*
//
```

The first three cards must contain the characters // in columns 1 and 2. The first of these cards is called a job card. In essence, this identifies your computer job and to what account, the usage of the computer is to be billed. The second cards specifies that you want to use the PL/I translator part of the operating system. This card also specifies certain options available to the user of the PL/I translator. The third card specifies that the input for the translator follows this card.

After these first three control cards, the PL/I program occurs. In OS/360, the program must be punched between columns 2 and 72. However, within these bounds, you are free to use any columns. Additional factors in the encoding of the program will be presented in the next chapter. After the last program card a /* card appears to identify the end of the program. The next control card which uses a //, specifies that the input data to be acquired by the program follows this card. After the input data which could be several cards, a /* is used to identify the end of the input data and the final // control card identifies the end of the job. Normally, this job would be one of many jobs to be processed by the computer, the jobs would all be collected together, loaded into the card reader input device and then processed. The control cards provide a convenient way of specifying what is to be done, identifying the beginning and ending of a computer job,/ and separating various components of the job, in this case, the PL/I program and its data.

An important point to remember is that the operating system and the PL/I translator are machine language programs. At various stages in the processing of the job, the machine language instructions of these programs are contained in the computer memory. We may visualize the step by step processing of the job in terms of a block diagram.


AFTER PREVIOUS JOB
OS/360 ACQUIRES INITIAL CONTROL CARDS
PL/I TRANSLATES THE PROGRAM INTO MACHINE LANGUAGE
THE TRANSLATED PROGRAM IS EXECUTED
OS/360 SKIPS TO THE NEXT JOB
NEXT JOB

STEP 1 Certain accounting information is noted and it is determined that the PL/I translator is to be executed.

STEP 2 The program statements as the input, are translated into machine language instructions. The /* card terminates the translation.

STEP 3 The translated program is executed, input values are acquired and output values created.

This type of processing in a computer system is referred to as batch processing. That is, several jobs are collected in a batch to be the input to the computer.

Let us take a closer look at the translation performed in STEP 2. This translation is analogous to gaining an understanding of some treatise written in a foreign language. That is, the treatise must first be translated into a language that is understood. If the treatise was written in Hungarian and was translated to English, Hungarian would be considered the source language (source of translation), and English would be the object language (object of translation). Likewise the encoded program written in the programming language is termed the source program or source language. The translator translates this source program into an object program or object language which is indeed the machine language program. The translation process is illustrated in Figure 2-2.

In the translation of one natural language to another, the translator is normally a person. However as we learned previously, the translator for a programming language translation is a machine language program which instructs the computer to accept the source program and to create a mahcine language program that reflects the instructions stated in the source program. When the translation to the object program is complete, the computer may then be instructed by the object program.

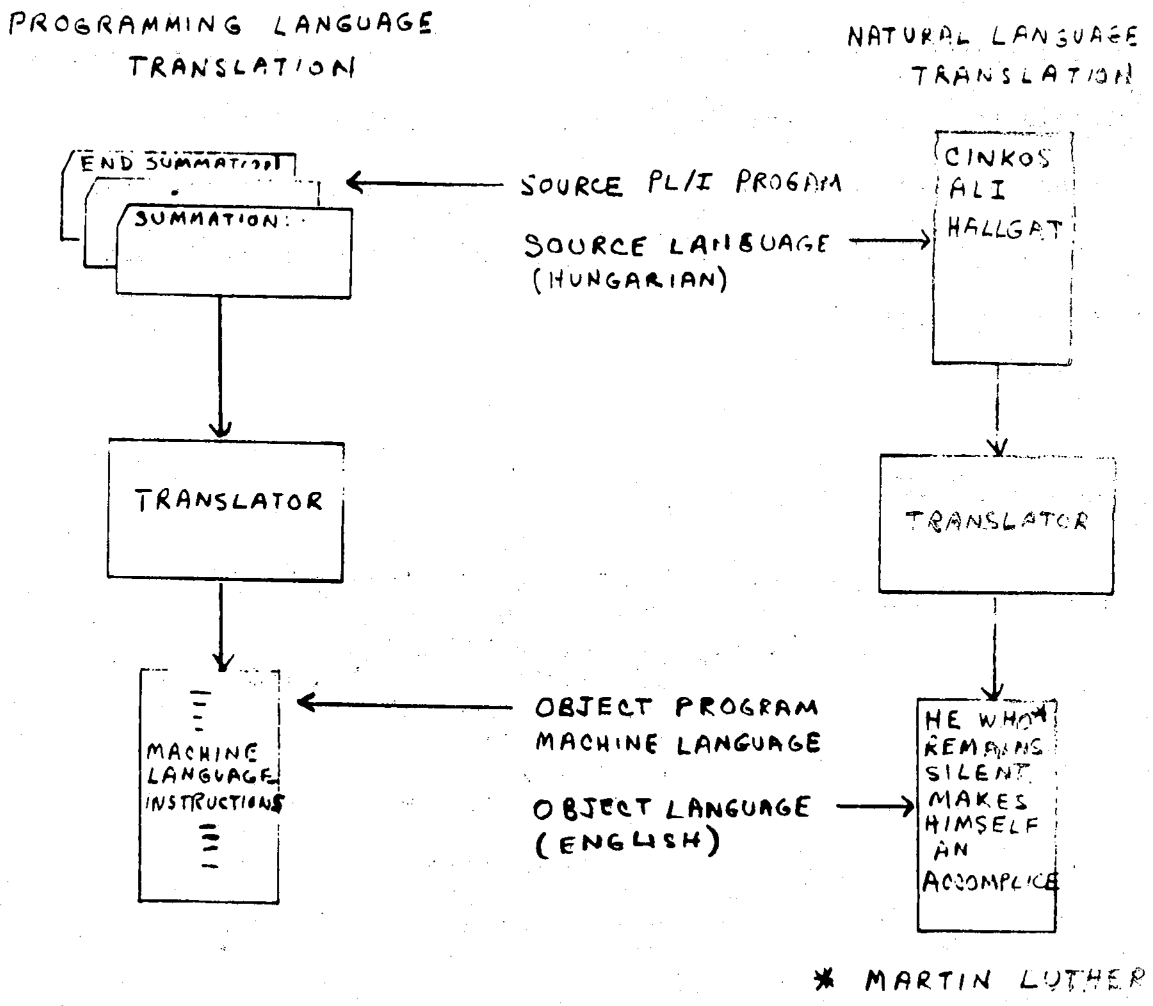

PROGRAMMING LANGUAGE TRANSLATION
NATURAL LANGUAGE TRANSLATION
END SUMMATION
SUMMATION:
SOURCE PL/I PROGRAM
SOURCE LANGUAGE (HUNGARIAN)
CINKOS ALI HALLGAT
TRANSLATOR
TRANSLATOR
MACHINE LANGUAGE INSTRUCTIONS
OBJECT PROGRAM MACHINE LANGUAGE
OBJECT LANGUAGE (ENGLISH)
HE WHO REMAINS SILENT MAKES HIMSELF AN ACCOMPLICE
* MARTIN LUTHER

Figure 2-2. Translation Process

The translator will normally create a listing via the printer output device of all of the statements of the program that it translates. In addition, it will diagnose certain types of errors in the program, and list these messages on the printer. These errors are normally due to the fact that the programmer did not follow the syntax rules for constructing PL/I programs.

There are a multitude of possible error messages that can be produced in a PL/I translation. These error messages are described in (3).

Assuming that our SUMMATION program was successfully translated and executed, we would obtain the following information in the listing produced on the computer.

PROGRAM LISTING

```
SUMMATION: PROCEDURE OPTIONS(MAIN);
           SUM,N = 0;
     NEXT: GET LIST(VALUE);
           IF VALUE = 0 THEN GOTO FINAL;
           SUM = SUM + VALUE;
           N = N + 1;
           GOTO NEXT;
    FINAL: PUT DATA(SUM,N);
           STOP;
           END SUMMATION;
```

EXECUTION RESULTS

```
SUM= 1.70000E+01          N=           2;
```

The notes to the right are not produced by the computer, they simply identify the origin of these components of the listing.

There would be additional information in the listing including the control cards and their information describing the various intermediate stages of processing the job. Based upon certain options specified in the control cards, various other information may be placed in the listing. Again, the reader should check the local control card conventions at the computer center where his jobs are processed.

You will notice in the output produced by the SUMMATION program that the value of SUM appears in floating-point decimal notation, whereas the value of N appears as a fixed-point decimal integer. The reason for these value formats will be described in the next chapter.

2-3 COMPUTER REPRESENTATION OF VALUES

An objective of most programming languages, including PL/I is to structure the language so that it is relatively machine independent. That is, given a program coded in PL/I for one computer, this same program may be processed on a different computer which has an implementation of PL/I. Complete machine independence is impossible due to the different ways of representing values in the computer memory of various computers. However, even though the data representation is a deterent to machine independence, the basic instructional steps of the PL/I program (i.e. statements) will in the majority of cases have the same or similar meaning on every computer in which PL/I is implemented. When using PL/I it is not essential to know all of the details of how data is manipulated in the computer. However, it is desirable to know the manner in which data values are represented and the relative efficiencies of processing various data values in the computer upon which your programs will be processed. This information will aid you in constructing more efficient PL/I programs. For this textbook, we are using the IBM

System/360, which several data representations. In chapter 3, we shall learn that one of the most important roles of the basic program activity of Value Declaration is to specify which data representations are to be used for program variables.

NUMBER SYSTEMS

In chapter 1, when we considered the SIMPLE computer and its memory, it was stated that all values in the memory are in the form of binary (i.e. base $_2$) digits (0 and 1), and by appropriately grouping binary digits, we can represent other number bases such as the familiar decimal base (i.e. base 10). In the following discussions of number bases, we shall consider decimal (base 10), binary (base_2), and hexadecimal (base_{16}) number systems. This later number base, hexadecimal, is simply a convenient grouping of binary digits. The System/360 permits the encoding of all three of these number bases. Since we shall be discussing values of three different number systems, the following notation will be used to explicitly state the base of a number.

$(v)_2$, $(v)_{10}$ and $(v)_{16}$ where v = the value

DECIMAL INTEGERS

Let us examine the manner in which decimal base 10 number is developed. For example, consider the following decimal value:

$$(2368)_{10}$$

We can state this value as the sum of successive powers of 10 multiplied by the digits of the number as follows:

$$\begin{aligned}
&= \underline{2 \times 10^3} + \underline{3 \times 10^2} + \underline{6 \times 10^1} + \underline{8 \times 10^0} \\
&= 2 \times 1000 + 3 \times 100 + 6 \times 10 + 8 \times 1 \\
&= 2000 + 300 + 60 + 8 = (2368)_{10}
\end{aligned}$$

where:	$10^0 = 1$	units position
	$10^1 = 10$	tens position
	$10^2 = 100$	hundreds position
	$10^3 = 1000$	thousands position

BINARY INTEGERS

The construction of a binary integer value is represented by a series of zero and one binary digits. As in the decimal number expansion, we can express the value of a binary number as a series of successive powers, however, we use the powers of 2 rather than the powers of 10 as in the decimal expansion. Consider the binary number:

$$(1101)_2$$

$$\begin{aligned} &= \underline{1 \times 2^3} + \underline{1 \times 2^2} + \underline{0 \times 2^1} + \underline{1 \times 2^0} \\ &= 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 \\ &= 8 + 4 + 0 + 1 = (13)_{10} \end{aligned}$$

where: $2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$

Consequently, the binary number $(1101)_2$ is equivalent to the decimal number $(13)_{10}$.

DECIMAL FRACTIONS

Next, let us consider the composition of a decimal fraction. A decimal fraction is the result of applying successive negative powers of 10. For example, consider the decimal fraction:

$$(.5921)_{10}$$

This may be expressed in powers of 10 as follows:

$$\underline{5 \times 10^{-1}} + \underline{9 \times 10^{-2}} + \underline{2 \times 10^{-3}} + \underline{1 \times 10^{-4}}$$

where: $10^{-1} = \frac{1}{10} = .1$

$10^{-2} = \frac{1}{100} = .01$

$10^{-3} = \frac{1}{1000} = .001$

$10^{-4} = \frac{1}{10000} = .0001$

Consequently, the value may be restated as:

$$5 \times .1 + 9 \times .01 + 2 \times .001 + 1 \times .0001 = \left(.5921\right)_{10}$$

BINARY FRACTIONS

A binary fraction may be specified in a similar manner. It is expressed as successive negative powers of 2. For example, consider the following binary fraction:

$$\left(.0111\right)_2$$

This may be expressed in powers of 2 as follows:

$$\underline{0 \times 2^{-1}} + \underline{1 \times 2^{-2}} + \underline{1 \times 2^{-3}} + \underline{1 \times 2^{-4}}$$

where $2^{-1} = \frac{1}{2} = .5$

$2^{-2} = \frac{1}{4} = .25$

$2^{-3} = \frac{1}{8} = .125$

$2^{-4} = \frac{1}{16} = .0625$

Consequently the value may be restated as:

$$\begin{array}{lllllllll} & 0 \times .5 & + & 1 \times .25 & + & 1 \times .125 & + & 1 \times .0625 & \\ = & 0 & + & .25 & + & .125 & + & .0625 & = \left(.4375\right)_{10} \end{array}$$

The binary fraction $\left(.0111\right)_2$ when expanded using the successive negative powers of 2 is equivalent to the decimal fraction $\left(.4375\right)_{10}$.

our knowledge of these conversions, consider expressing a binary value which has a whole part and a fraction. Consider the binary value:

$$\left(1011.\ 101\right)_2$$

This may be stated in powers of 2 as follows:

$$\begin{array}{lllllllllllll} \underline{1 \times 2^3} & + & \underline{0 \times 2^2} & + & \underline{1 \times 2^1} & + & \underline{1 \times 2^0} & + & \underline{1 \times 2^{-1}} & + & \underline{0 \times 2^{-2}} & + & \underline{1 \times 2^{-3}} \\ = \underline{1} \times 8 & + & \underline{0} \times 4 & + & \underline{1} \times 2 & + & \underline{1} \times 1 & + & \underline{1} \times .5 & + & \underline{0} \times .25 & + & \underline{1} \times .125 \\ = 8 & + & 0 & + & 2 & + & 1 & + & .5 & + & 0 & + & .125 \\ = 11 & + & .625 & = & \left(11.625\right)_{10} & & & & & & & & \end{array}$$