Міністерство освіти і науки України
Запорізький національний технічний університет

МЕТОДИЧНІ ВКАЗІВКИ

**до практичних занять з англійської мови
для студентів
спеціальностей «Інженерія програмного забезпечення»
(121.1, 121.2), «Комп'ютерні науки та інформаційні технології
проектування» (122.1)
та «Комп'ютерна інженерія» (123.1, 123.2)
денної форми навчання**

2018 р.

Методичні вказівки до практичних занять з англійської мови для студентів спеціальностей «Інженерія програмного забезпечення» (121.1, 121.2), «Комп'ютерні науки та інформаційні технології проектування» (122.1) та «Комп'ютерна інженерія» (123.1, 123.2) денної форми навчання. / Укл. В.Г. Кузьменко. – Запоріжжя: ЗНТУ, 2018. – 66 с.

Укладач: В.Г. Кузьменко
Рецензент: Ю.А. Соболь, доц., канд. філол. наук
Експерт: Р.К. Кудерметов, доц. канд. тех. наук
С.К. Корнієнко, доц. канд. тех. наук
Відповідальний за випуск: В.Г. Кузьменко

Затверджено
на засіданні кафедри
іноземних мов
Протокол № 4 від 27.03.18

Рекомендовано до видання
НМК КНТ факультету
Протокол № 9 від 26.04.18

CONTENTS

UNIT 1. PROGRAMMING LANGUAGES AND SOFTWARE CONCEPTS

Task I. Before reading the text translate the words and word combinations and match them with the appropriate explanation.

1.Applications software	– a)translates the instructions of a high-level language to a machine-language instructions that the computer can interpret and execute.
2.Systems software	– b)a family of systems software programs supplied by the computer system vendor which minimize turnaround time, maximize throughput and optimize the use of computer resources.
3.Machine language	– c)controls the flow of data to and from remote locations, i.e. prepare data for transmission, pole remote terminals for input, establish the connection between two terminals, encode and decode data, and perform parity checking.
4.Assembler language	– d)the actual high-level programming language instructions.
5.Compiler	– e)provides the interface between application programs and the data base.
6.Source program	– f)use high-level Englishlike instructions to retrieve and format data for user inquiries and reporting. a programmer need only specify “what to do”, not “how to do it”.

7.Object program – g)simultaneous execution of more than one program at a time on a single computer.

8.Interpreter – h)the output of the compilation process, it resides in primary storage and can be executed upon your command.

9.Problem-oriented languages – i)when two or more programs are competing for the printer both programs are executed and the printer output for one is temporarily loaded to magnetic disk. as the printer becomes available, the output is called from magnetic disk.

10.Procedure-oriented languages j)assist in the development of full scale information systems during which programmers specify through an interactive dialog with the system what information processing tasks are to be performed.

11.Query languages – k)designed to perform certain personal, business or scientific processing tasks.

12.Application generator – l)assigns a primary storage address to each byte of the object program.

13.Operating system – m)a system software addition to the operating system that effectively expands the capacity of primary storage through the use of software and secondary storage.

14.Supervisor – n)uses easily recognized symbols, called mnemonic's to represent instructions.

15.Linkage editor – o)permits programmers to model

any scientific or business procedure.

16.Utility programs – p)designed for a particular application.

17.Communications software – q)loads operating system and applications programs to primary storage as they are needed.

18.Data base management system software – r)mare general and supports the basic functions of the computer.

19.Multiprogramming – s)eliminate the need for user to write a program every time they need to perform certain computer operations.

20.Spooling – t)translates and executes each source program instruction before translating and executing the next one.

21.Virtual memory – u)the only language that can be executed on a particular computer.

1.1. Programming and Software

Most of us will agree that computers have become an integral part of society.

We can touch them and see the results of their seemingly endless capabilities. But the computer-literate person recognizes that hardware is useless without software, and software is useless without hardware.

A computer system does nothing until directed to do so. A program, which consist of instructions to the computer, is the means by which we tell a computer to perform certain operation. These instructions are logically sequenced and assembled through the act of programming. Programmers use a variety of programming languages, such as COBOL and BASIC, to communicate instructions to the computer. We use the term software to refer to the programs that direct the activities of the computer system. Software falls into two general categories: applications and system. Applications software is designed and written to perform specific personal, business, or scientific processing or statistical analysis. Other examples

include the programs for claims processing (insurance), tax collection (manufacturing), and satellite trajectory tables (NASA).

System software is more general than applications software and is usually independent of any specific application area. System software programs support all application software by directing the basic function of the computer. For example, when the computer is turned on, an initialization program prepares and readies all devices for processing. Software that permits us to write programs in COBOL and BASIC is also system software. The operating system, too, discussed later in this chapter, is classified as system software.

1.2. Generations of Programming Languages

We "talk" to computers within the framework of a particular programming language. There are many different programming languages, most of which have highly structured sets of rules. The selection of a programming language depends on who is involved and the nature of "conversation". The president of a company may prefer a different type of language then a professional programmer; language used for payroll processing may not be appropriate for ad-hoc (one time) inquiries.

Like computers, programming languages have evolved in generations. With each new generations, fewer instructions are needed to instruct the computer to perform a particular task. That is, a program written in a first-generation language that computes and prints student grade averages may require 100 or more instructions; the same program in a fourth-generation languages may have fewer then ten instructions.

The hierarchy of programming languages, shown in Figure 1.1, illustrates the relationships between the six generations of programming languages. The later generation do not necessarily provide us with greater programming capabilities, but they do provide a more sophisticated programmer/computer interaction. In short, each new generation is easier to understand and use. For example, in the fourth, fifth, and sixth generation, we need only instruct the computer system what to do, not necessarily how to do it.

The ease with which the later generation can be used is certainly appealing, but the earlier languages also have their advantages. All six generations of languages are in use today.

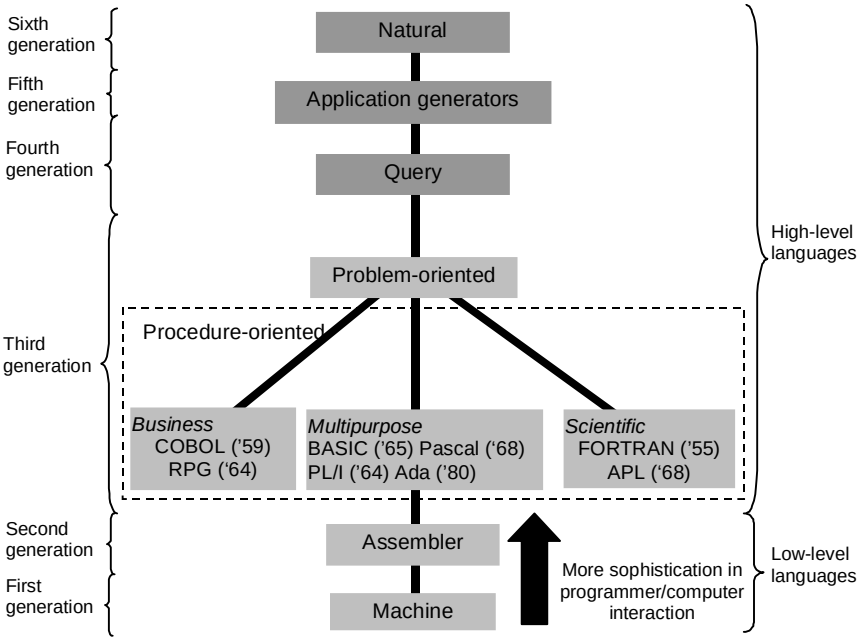


Figure 1.1 – The Hierarchy of Programming Languages. As you progress from one generation of programming languages to the next, fewer instructions are required to perform a particular programming task.

1.3. The First and Second Generations: "Low–Level"

Machine Language

Each computer has only one programming language that can be executed- the machine language. We talk of programming in COBOL, Pascal, and BASIC, but all of these languages must be translated to the machine language of the computer on which the program is to be executed. These and other high-level languages are simply a convenience for the programmer.

Machine-language programs, the first generation, are written at the most basic level of computer operation. Because their instructions are written at the most basic level of operation, machine language and assembler language (see below) are collectively called low-level languages. In machine language, instructions are coded as a series of 1s and 0s. As you

might expect, machine-language programs are cumbersome and difficult to write. Early programmers had no alternative. Fortunately, we do.

Assembler Language

A set of instructions for an assembler language is essentially one-to-one with those of machine language. Like machine languages, assembler languages are unique to a particular computer. The big difference between the two types is the way the instructions are represented by the programmer. Rather then a cumbersome series of 1s and 0s, assembler languages use easily recognized symbols, called mnemonics, to represent instructions (see Figure 1.2). For example, most assembler languages use the mnemonic "A" to represent an "Add" instruction. The assembler languages ushered in the second generation of programming languages.

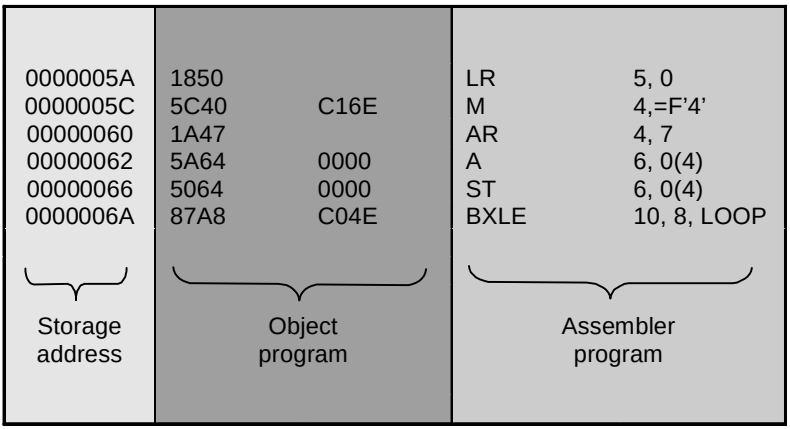


Figure 1.2 – Part of an Assembler-Language Program. Even relatively simple assembler-language programs may contain a hundred or more instructions. The machine-language equivalent (object program) is show in hexadecimal. The leftmost column contains the primary storage addresses for each machine-language and assembler-language instruction

Prior to 1970, machine- and assembler-level languages were often used for applications program development. Although it took longer, many programmers preferred assembler programming because they believed it used the computer system more efficiently. Since then, the power and flexibility of new generations of languages have put them beyond low-level languages in terms of both human and computer efficiency. Consequently,

most programming is now done in high-level languages (third through sixth generations).

1.4. Compilers and Interpreters: Programs for Programs

No matter which language a high-level program is written in, it must be translated to machine language before it can be executed. This conversion of high-level instructions is done by systems software programs called compilers and interpreters.

Compilers

The compiler program translates the instructions of high-level language, such as COBOL, to machine-language instructions that the computer can interpret and execute. A separate compiler (or an interpreter, discussed in the next section) is required for each programming language intended for use on a particular computer system. That is, to execute both COBOL and Pascal programs, you must have a COBOL compiler and Pascal compiler. High-level programming languages are simply a programmer convenience; they cannot be executed in their source, or original, form.

The actual high-level programming-language instructions, called the source program, are translated or compiled to machine-language instructions by a compiler. The circled numbers in Figure 1.3 cross-reference the following numbered discussion of the compilation process.

1. Suppose you want to write a COBOL program. You first enter the instructions into the computer system through an on-line workstation. Having done so, you identify the language (COBOL) in which you wrote the program and request that the program be compiled.

2. The COBOL compiler program is called from secondary storage and loaded to primary storage along with the COBOL source program. [Note: Step 3 will be attempted but not completed if the source program has errors or bugs (see Step 4).]

3. The COBOL compiler translates the source program to a machine language program called an object program. The object program is the output of the compilation process. At this point, the object program resides in primary storage and can be executed upon your command.

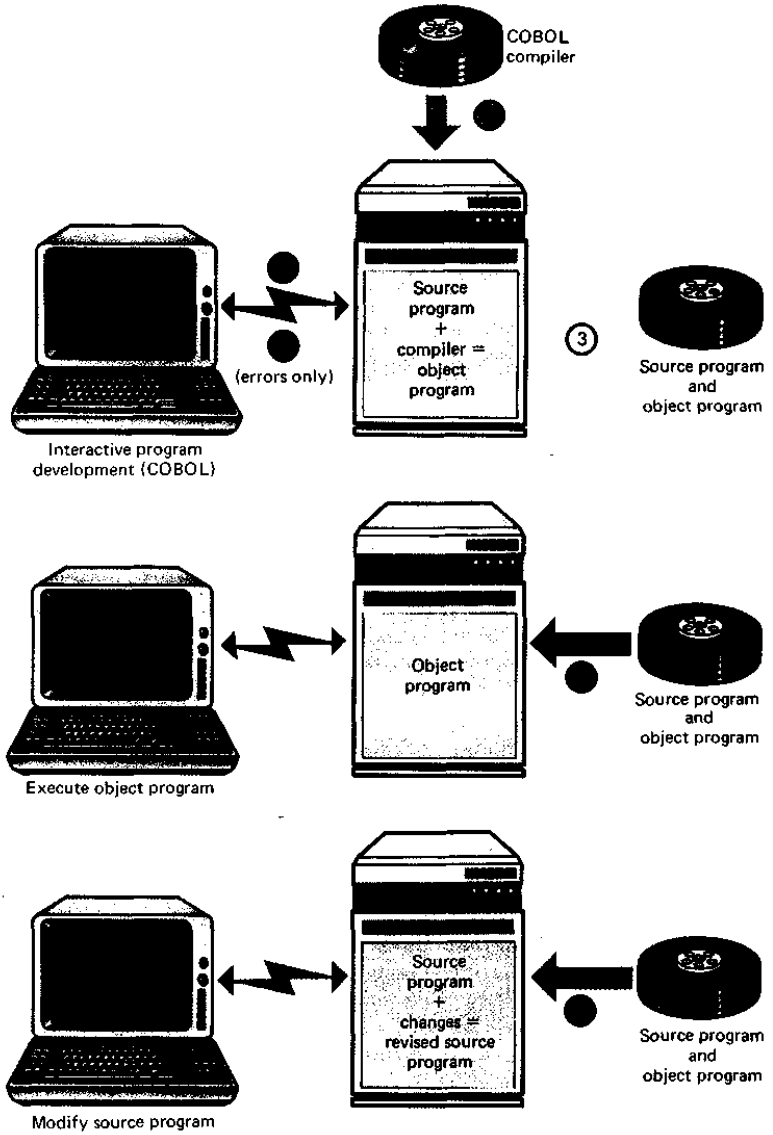


Figure 1.3 – The Compilation Process. A source program is translated to an object program for execution, Steps of the compilation process are discussed in the text

The compilation process can be time consuming, especially for large programs. Therefore, if you intend to execute the program at a later time, perhaps during another session, you should store the object program on secondary storage for later recall. On most mainframe computer systems this is done automatically.

4. If the source program contains a syntax error (e.g., an invalid instruction format), the compiler will display an error message, or diagnostic, on the workstation screen, then terminate the compilation process. A diagnostic identifies the program statement or statements in error and the cause of the error. Syntax errors usually involve invalid instructions. Consider the following COBOL statement: DISPLAY "WHOOPS". The statement is invalid because DISPLAY is misspelled.

As a programmer, you will make the necessary corrections and attempt the compilation over, and over, and over again, until the program compiles and executes. Don't be discouraged. Very few programs compile on the first, second, or even third attempts. When your program finally compiles and executes, don't be surprised if the output is not what you expected. A "clean", or error-free, compilation is likely to surface undetected logic errors. For example, your program logic might result in an attempted division by zero; this is mathematically and logically impossible and will result in a program error. In most cases, you will need to remove a few such bugs in the program logic and in the I/O formats before the program is finished.

5. Suppose you come back the next day and wish to execute your COBOL program again. Instead of repeating the compilation process of Step 2, you simply call the object program from secondary and load it to primary storage for execution. Since the object program is already in machine language, no compilation is necessary.

6. If you want to make any changes to the original source programs, you will: recall the original source code from secondary storage, make the changes, recompile the program, and create an updated object program (repeat Steps1-4).

Programs that are run frequently are stored and executed as object programs. Recompilation is necessary only when the program is modified.

Interpreters

An interpreter is a system software program that ultimately performs the same function as a compiler–but in a different manner. Instead of

translating the entire source program in a single pass, an interpreter translates and executes each source program instruction before translating and executing the next. The obvious advantage of interpreters over compilers is that an error in instruction syntax is brought to the attention of the programmer immediately, thereby prompting programmer to make corrections during program development. This is a tremendous help.

As we know, advantages are usually accompanied by disadvantages. The disadvantage of interpreters is that they do not use computing resources as efficiently as a program that has been compiled. Since the interpreter does not produce an object program, it must perform the translation process each time a program is executed.

For programs that are to be run often, programmers take advantage of the strengths of both interpreters and compilers. First, they develop and debug their programs using an interpreter. Then, they compile the finished program to create a more efficient object program that can be used for routine processing.

1.5. The Third Generation: for Programmer Convenience

A quantum leap in programmer convenience accompanied the introduction of the third generation of programming languages. A third-generation language is placed in one of two categories: procedure-oriented languages or problem-oriented languages (review Figure 1.1).

Procedure-Oriented Languages

The flexibility of procedure-oriented languages permits programmers to model almost any scientific or business procedure. Instructions are coded, or written, sequentially and processed according to program specifications.

Unless triggered by program logic to do otherwise, the processor selects and executes instructions in the sequence in which they are written. In a production payroll system, for example, a particular sequence of program instructions is executed for salaried employees; another sequence is executed for hourly employees. The same sequence of program instructions is repeated for each salaried employee, however, and the other sequence of instructions is repeated for each hourly employee.

Procedure-oriented languages are classified as scientific, business, or multipurpose. These are illustrated in Figure 1.1 and discussed below.

Scientific Languages. Scientific languages are algebraic/formula-type languages. These are specifically designed to meet typical scientific processing requirements, such as matrix manipulation, precision calculations, iterative processing, the expression and resolution of mathematical equations, and so on. For example, engineers and actuaries turn to scientific languages when writing programs for statistical analysis.

FORTRAN. FORTRAN (Formula Translator), the first procedure-oriented language, was developed in 1955. It was, and it remains, the most popular scientific language (see Figure 1.4). However, the rise in the popularity of BASIC, Pascal, APL, and PL/I has begun to chip away at FORTRAN's hold on first place.

```
00100          INTEGER EOQ
00200    10    TYPE 100
00300    100   FORMAT(/,1X, 'ЙNTER COST/ITEM, NO. REQUIRED/YEAR, AND ITEM',
00400          1 'STORAGE COST')
00500          ACCEPT*, COST, NUMBER, STCOST
00600          EOQ=SQRT ( (2*COST*NUMBER)/STCOST)
00700          TYPE 200, EOQ
00800    200   FORMAT(/, 1X, 'THE ECONOMIC ORDER QUANTUTY IS ', 16)
00900          TYPE 300
01000    300   FORMAT (/, 1X, 'DO YOU WISH TO CALCULATE ANOTHER EOQ? (Y OR N)')
01100          ACCEPT 400, MORE
01200    400   FORMAT(A1)
01300          IF (MORE.EQ. 'Э') THEN GO TO 10
01400          END
```

```
ENTER COST/ITEM, NO. REQUIRED/YEAR, AND ITEM STIRAGE COST
?22.5, 5000, 2.15

THE ECONOMIC ORDER QUANTITY IS 323

DO YOU WISH TO CALCULATE ANOTHER EOQ? (Y OR N)
? N
```

Figure 1.4 – A FORTRAN Program. This program computers the economic order quantity (EOQ)

APL. APL (A Programming Language) is a symbolic interactive programming language used primarily by engineers, mathematicians, and scientists. APL, introduced in 1968, is unique in that a special workstation is required for writing APL programs. Its keyboard has a special character set that includes the symbols for writing APL instructions. These symbols

provide programmers with a “shorthand” that speeds up the process of coding a program.

APL’s limited-output formatting capability has not concerned engineers and scientists. Their concern is primarily the solution and not the manner in which the information is displayed (see Figure 1.5).

```
      ∇AVG[⎕]∇
      ∇ AVG QUIZSCORES;NRSCORES;MEAN
[1]   ⍝   DESCRIPTION OF FUNCTION 'AVERAGE'
[2]   ⍝      ACCEPTS ANY NUMBER OF QUIZ SCORES THEN COMPUTES AND DISPLAYS
[3]   ⍝      THE AVERAGE ROUNDED TO ONE DECIMAL PLACE
[4]   NRSCORES←⍴,QUIZSCORES
[5]   MEAN←(+/,QUIZSCORES)÷NRSCORES
[6]   'THE AVERAGE OFTHE',(⍕ NRSCORES),' QUIZ SCORES IS ',(5 1⍕ MEAN)
      ∇
```

```
AVG 94 83 88

THE AVERAGE OF THE 3 QUIZ SCORES IS 88.3
```

Figure 1.5 – An APL Program. This program accepts any number of quiz scores, then computes and displays the average, which is rounded to one decimal place

Business Languages. Business programming languages are designed to be effective tools for developing business information systems. The strength of business-oriented languages lies in their ability to store, retrieve, and manipulate alphanumeric data.

The arithmetic requirements of most business systems are minimal. Although sophisticated mathematical manipulation is possible, it is cumbersome to achieve, so it is best left to scientific languages.

COBOL. COBOL, the first business programming language, was introduced in 1959. It remains the most popular. The original intent of the developers of COBOL (Common Business Oriented Language) was to make its instructions approximate the English language. Although COBOL is a very powerful language, many programmers consider it to be excessively wordy. Here is a typical COBOL sentence: “IF SALARY-CODE IS EQUAL TO 'H' MULTIPLY SALARY BY HOURLY-RATE GIVING GROSS-PAY ELSE PERFORM SALARIED-EMPLOYEE-

ROUTINE". Note that the sentence contains several instructions and even a period.

Professional programmers have a love/hate relationship with COBOL. It is not uncommon for one programmer's opinion of COBOL to be 180 degrees out of phase with that of a colleague. Personal opinions notwithstanding, COBOL has over 20 years of momentum and acceptance, and it will probably be a mainstay of business systems programming for years to come. Over half of all production programs for currently running business systems are written in COBOL.

The American National Standards (ANS) Institute has established standards for COBOL and other languages. The purpose of these standards is to make COBOL programs portable. A program is said to be portable if it can be run on a variety of computers. Unfortunately, the ANS standards are only casually followed; consequently, it is unlikely that a COBOL program written for a Burroughs computer, for example, can be executed on a Honey-well computer without some modification.

COBOL programs are divided into the following four divisions (see Figure 1.6):

♦ Identifications Division. Identifies general information, such as program name, programmer, date written, and so on.

♦ Environment Division. Identifies the type of computer system (manufacturer and model) on which the program is to be run as well as the data storage and I/O devices to be used.

♦ Data Division. Describes the format of all data elements, internal variables, records, and files used by the program.

♦ Procedure Division. Contains the logic of the program and the sequence of instructions to be executed. Within the procedure division, instructions are written as sentences and logically combined to from paragraphs.

RPG. RPG (Report Program Generator) was originally developed in 1964 for IBM's entry-level punched-card business computers and for the express purpose of generating reports. As punched cards went the way of vacuum tubes, RPG remained–evolving from a special-purpose problem-oriented language to a general-purpose procedure-oriented language. Its name has made RPG the most misunderstood of the programming languages. People who do not know RPG still associate it with report

generation when, in fact, it has become a powerful programming language that matured with the demands of RPG users.

```
0100   IDENTIFICATON DIVISION.
0200   PROGRAM–ID.              PAYPROG.
0300   REMARKS.                 PROGRAM TO COMPUTE GROSS PAY.
0400   ENVIRONMENT DIVISION.
0500   DATA DIVISION.
0600   WORKING–STORAGE SECTION.
0700   01 PAY–DATA.
0800            05 HOURS          PIC 99V99.
0900            05 RATE           PIC 99V99
1000            05 PAY            PIC 9999V99
1100   01 LINE–1.
1200            03 FILLER         PIC X(5)        VALUE SPACES.
1300            03 FILLER         PIC X(12)       VALUE "GROSS PAY IS ".
1400            03 GROSS-PAY      PIC $$$9.99.
1500   01 PRINT–LINE             PIC X(27).
1600   PROCEDURE DIVISION.
1700   MAINLINE–PROCEDURE.
1800            PERFORM ENTER–PAY.
1900            PERFORM COMPUTE–PAY.
2000            PERFORM PRINT–PAY.
2100            STOP RUN.
2200   ENTER–PAY.
2300            DISPLAY "ENTER HOURS AND RATE OF PAY".
2400            ACCEPT HOURS, RATE.
2500   COMPUTE–PAY.
2600            MULTIPLY HOURS BY RATE GIVING PAY ROUNDED.
2700   PRINT–PAY.
2800            MOVE PAY TO GROSS–PAY.
2900            MOVE LINE–1 TO PRINT–LINE.
3000            DISPLAY PRINT–LINE.
```

```
Enter hours and rate of pay
43, 8.25
          Gross pay is $354.75
```

Figure 1.6 – A COBOL Program. This program computes gross pay for hourly wage earners

RPG has always differed somewhat from other procedure-oriented languages in that the programmer specifies certain processing requirements by selecting the desired programming options. That is, during a programming session, the programmer is presented with prompting formats at the bottom of the workstation screen. The programmer requests the

prompts for a particular type of instruction, then responds with the desired programming specifications (see Figure 1.7).

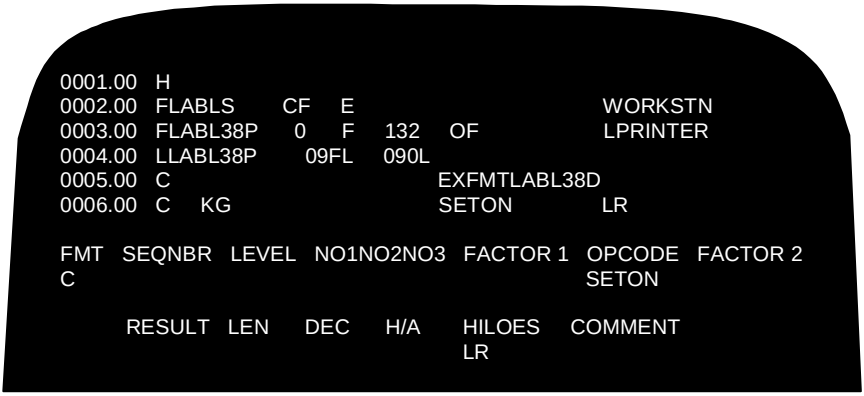


Figure 1.7 – Interactive RPG Programming. Show here is an interactive specification menu used in the preparation of an RPG program

Smaller computer installations may use RPG exclusively as the programming language for all their information processing needs. In larger installations, RPG is still used primarily for the preparation of reports.

Multipurpose Languages. Multipurpose languages are equally effective for both business and scientific applications. These languages are an outgrowth of the need to simplify the programming environment by providing programmers with one language that is capable of addressing all of a company's programming needs.

BASIC. BASIC, developed in 1964, is the primary language supported by millions of personal computers. BASIC is also used extensively on mainframe computer systems, primarily for one-time "quick and dirty" programs.

BASIC is perhaps the easiest language to learn and use (see Figure 1.8). It is commonly used for both scientific and business applications – and ever for developing video games. The widespread use of BASIC attests to the versatility of its features. In fact, BASIC is the only programming language that is supported on virtually every computer.

PL/I. PL/I (Programming Language/I), introduced in 1964, was hailed as the answer to many of the shortcomings of existing programming languages, such as COBOL and FORTRAN. It has not, however, won the widespread acceptance that was originally anticipated. The slow acceptance of PL/I is due, not to lack of quality, but to the substantial investment and

commitment that companies already had in COBOL and FORTRAN. A PL/I example is shown in Figure 1.9.

Pascal. During the last decade Pascal, named after the seventeenth-century French mathematician Blaise Pascal, has experienced tremendous growth. Introduced in 1968, Pascal is considered the state of the art among widely used procedure-oriented languages (see Figure 1.10).

Although only 1% to 2% of business-system programs are now written in Pascal, its power, flexibility, and self-documenting structure are factors that cannot be overlooked for long. Many college and university computer science and information systems programs are advocating Pascal. With students graduating and carrying their expertise into the business world by the thousands, it stands to reason that in the future, Pascal will attain growing acceptance in the business community.

C. The results of a recent employment survey showed C programmers to be in the greatest demand. Developers of proprietary software are very interested in C because it is considered more transportable than other languages. That is, it is relatively machine independent: a C program written for one type of computer can be run on another type with little or no modification. When faced with time-comsuming and expensive task of translating programs to run on a dozen micros and mainframes, C becomes an inviting option.

Another inviting quality of C is that it can be used for developing both systems and applications software. Traditionally, system software has been written in an assembler language and applications software in a procedure-oriented language.

Ada. Ada is multipurpose language developed for the U.S. Department of Defence. The language was named to honour the nineteenth-century pioneer, Lady Augusta Ada Lovelace, considered by some to be the first programmer. Although Ada has been selected to replace COBOL as a standard for the U.S. Department of Defence, relatively few programmers know and understand Ada. Its developers are, however, optimistic that as more people begin to study it, Ada will gain momentum and widespread acceptance not only in the military but in the private sector as well. Ada is the most recent and, perhaps, the most sophisticated procedure-oriented language.

```
100    REM   <<<<Start Main >>>>
110    REM
120    REM   Call Module 1.1 – Accept Number
130    GOSUB   1000
140    REM   Begin loop to be executed 'NUMBER' times
150    COUNT = 1
160    WHILE COUNT <= NUMBER
170        REM   Call Module 1.2 – Accept Grades
180        GOSUB   2000
190        REM   Call Module 1.3 – Compute Average
200        GOSUB   3000
210        REM   Call Module 1.4 – Display Results
220        GOSUB   4000
230    WEND
240    END
250    REM   <<<<End Main>>>>
999    REM
1000   REM   ==== Module 1.1 – Accept Number
1010   PRINT "Enter total number of students";
1020   INPUT NUMBER
1030   PRINT
1999   RETURN
2000   REM   ==== Module 1.2 – Accept Grades
2010   REM   Input student grades and add one to loop counter 'COUNT'
2020   PRINT   "Enter three quiz scores separated by commas";
2030   INPUT  Q1,   Q2,   Q3
2040   REM   increment loop counter
2050   COUNT = COUNT + 1
2999   RETURN
3000   REM   ==== Module 1.3 – Compute Average
3010   AVERAGE = (Q1 + Q2 + Q3) / 3
3999   RETURN
4000   REM   ==== Module 1.4 – Display Results
4010   PRINT   "The average grade is ";   AVERAGE
4020   PRINT
4999   RETURN
```

```
run
Enter total number of students?   3

Enter three quiz scores separated by commas?   73,91,85
The average grade is   83

Enter three quiz scores separated by commas?   86,88,99
The average grade is   91

Enter three quiz scores separated by commas?   66,84,75
The average grade is   75

Ok
```

Figure 1.8 – A BASIC Program. This program average three quiz scores for any number of students

```
0001   FUTVAL:   PROCEDURE OPTIONS (MAIN);
0002   DECLARE (INVESTMENT, FUTUREVALUE) FIXED DECIMAL (7,2);
0003   DECLARE (YEARS, INTEREST) FIXED DECIMAL (2,1), DECLARE DUMMY CHAR(72);
0004   DISPLAY  ('ENTER INITIAL INVESTMENT ') REPLY(DUMMY);
0005   INVESTMENT = DUMMY;
0006   DISPLAY  ('ЙNTER NUMBER OF YEARS ') REPLY(DUMMY);
0007   YEARS = DUMMY;
0008   DISPLAY  ('ENTER INTEREST RATE ') REPLY(DUMMY);
0009   INTEREST = DUMMY;
0010   FUTUREVALUE = INVESTMENT*(1.0 + INTEREST/100.) **YEARS;
0011   DISPLAY  ('THE FUTURE VALUE OF $'  INVESTMENT  '  COMPOUNDED');
0012   DISPLAY  ('ANNUALLY FOR  '  YEARS  '  YEARS AT  '  INTEREST  '% IS  $'
0013     FUTUREVALUE;
0014   END;
```

```
ENTER INITIAL INVESTMENT
5500
ENTER NUMBER OF YEARS
5
ENTER INTEREST RATE
12.5
THE FUTURE VALUE OF $ 5500.00 COMPOUNDED
ANNUALLY FOR. 5.0 YEARS AT 12.5% IS $ 9911.18
```

Figure 1.9 – A PL/I Program. This program computes the future value of an investment, given the interest rate and the number of years the investment is held

Problem-Oriented Languages

A problem-oriented language is designed to address a particular application area or to solve a particular set of problems. Problem-oriented languages do not require the programming detail of procedure-oriented ones. The emphasis of problem-oriented languages is more on input and the desired output than on the procedures or mathematics involved.

Problem-oriented languages have been designed for scores of applications: simulation (e.g., GPSS, SLAM); programming machine tools (e.g., APT); analysis of stress points in buildings and bridges (e.g. COGO); and word processing (e.g., Wordstar).

Although simulation problems can be programmed in FORTRAN, Pascal, and other procedure-oriented languages, the same problems can be programmed much more easily in SLAM, a problem-oriented language. Problem-oriented languages do not have the flexibility of procedure-

oriented languages; they are limited to the application for which they are designed.

```
 1 PROGRAM MEANDEV (INPUT, OUTPUT);
 2 CONST
 3   MAX = 100;    YES = 'Y';    NO = 'N';    BLANK = '  ';
 4 VAR
 5   COUNT, I: INTEGER;   VALUE: ARRAY [1. .MAX] OF REAL
 6   MEAN, SUM. SD. SDEVIATION, NUM: REAL;   RESPONSE: CHAR;
 7 BEGIN
 8   WRITELN('COMPUTE MEAN AND STANDARD DEVIATION FOR UP TO 100 VALUES');
 9   WRITELN;
10   REPEAT
11     WRITELN('ENTER OBSERVATION VALUES.   END INPUT WITH -999.9');
12     AD : = 0.0;   COUNT : = 0;   SUM : 0.0;              (* INITIALIZE VARIABLES *)
13     READ(NUM);                                          (* READ IN VALUES *)
14     WHELE(NUM <> -999.9) AND (COUNT < MAX) DO BEGIN
15       COUNT : = COUNT + 1;
16       VALUE[COUNT] : = NUM;
17       SUM : = SUM + NUM;
18       READ(NUM)
19     END;                                                (* WHILE *)
20     IF COUNT > 0 THEN
21       MEAN : = SUM/COUNT;                               (* COMPUTE MEAN *)
22     FOR I: = 1 TO COUNT DO
23       SD : = SD + SQR(VALUE[I] – MEAN);
24     SDEVIATION : = SQRT(1 / (COUNT – 1) * SD);   (* COMPUTE STANDARD DEVIATION *)
25     WRITELN('NUMBER IN SAMPLE = ', COUNT: 1);
26     WRINELN('MEAN = ', MEAN: 6: 2);
27     WRIYELN('STANDARD DEVIATION = ', SDEVIATION: 6: 5);
28     REPEAT
29       WRITELN('ANOTHER SAMPLE (Y  OR  N)?');
30       REPEAT                                            (* READ PAST LEADING BLANKS *)
31         READ(RESPONSE)
32       UNTIL RESPONSE <> BLANK
33     UNTIL (RESPONSE = YES) OR (RESPONSE = NO)
34   UNTIL RESPONSE = NO
35 END                                                     (* PROGRAM *)
```

```
COMPUTE MEAN AND STANDARD DEVIATION FOR UP TO 100 VALUES

ENTER OBSERVATION VALUES. END INPUT WITH –999.9
128  99.2  136.4  114  108.3  100.6  92.8  124  -999.9
NUMBER IN SAMPLE = 8
MEAN = 112.91
STANDARD DEVIATION = 15.44034
ANOTHER SAMPLE (Y or N)?
N
```

Figure 1.10 – A Pascal Program. This program computes the mean and standard deviation of any number of observations

1.6. The Fourth Generation: Query Languages

Query Languages

Over the years, most organizations accumulated large amounts of computer-based data that only the professional programmer could access. Prior to fourth-generation languages (mid-1970's), users had to describe their information needs to a professional programmer, who would then use procedure-oriented languages to supply the information. Fulfilling a typical user request would take at least a couple of days and as much as two weeks. By then, the information might no longer be timely. With fourth-generation query languages, these same ad-hoc requests, or queries, can be completed in minutes by the user, without involving computer professionals!

Principles and Use. Query languages use high-level Englishlike instructions to retrieve and format data for user inquiries and reporting. Most of the procedure portion of a query-language program is generated automatically by the computer and the language software. As a programmer in query languages, you need only specify "what to do" and "how to do it". Thus, the use of query languages for certain information needs yields substantial improvements in programming productivity. The features of a query language include Englishlike instructions, limited mathematical manipulation of data, automatic report formatting, sequencing (sorting), and record selection by criteria.

With four to eight hours of training and practice, you can learn to write query-language programs, make inquiries, and get reports-without the assistance of a professional programmer. With query languages, it may be easier and quicker to sit down at the nearest workstation and write a program than it is to relate inquiry or report specifications to a programmer. Now managers can attend to their own seemingly endless numbers of ad-hoc requests. Query languages benefit everyone concerned. Users get the information that they need quickly, and programmers can devote their time to an ever-increasing backlog of information system projects. Professional programmers use query languages to increase their productivity as well.

Query-Language Example. The query-language example presented here will give you a feel for the difference between a procedure-oriented language, such as COBOL, and a query language. About 20 query languages are commercially available, such as NOMAD2 and EASYTRIEVE Plus. The example shows how a representative query language can be used to generate a management report. Suppose a

personnel manager wanted the report shown in Figure 1.11. To obtain the report, the manager wrote the query-language program in Figure 1.12:

PAYROLL DEPARTMENTS 911, 914

DEPARTMENT	EMPLOYEE NAME	EMPLOYEE NUMBER	SEX	NET PAY	GROSS PAY
911	ARNOLD	01963	1	356.87	445.50
911	LARSON	11357	2	215.47	283.92
911	POWELL	11710	1	167.96	243.20
911	POST	00445	1	206.60	292.00
911	KRUSE	03571	2	182.09	242.40
911	SMOTH	01730	1	202.43	315.20
911	GREEN	12829	1	238.04	365.60
911	ISAAC	12641	1	219.91	313.60
911	STRIDE	03890	1	272.53	386.40
911	REYNOLDS	05805	2	134.03	174.15
911	YOUNG	04589	1	229.69	313.60
911	HAFER	09764	2	96.64	121.95
DEPARTMENT TOTAL				2,522.26	3,497.52
914	MANHART	11602	1	250.89	344.80
914	VETTER	01895	1	189.06	279.36
914	GRECO	07231	1	685.23	1,004.00
914	CROCI	08262	1	215.95	376.00
914	RYAN	10961	1	291.70	399.20
DEPARTMENT TOTAL				1,632.83	2,403.36
FINAL TOTAL 17 RECORDS TOTALED				4,155.09	5,900.88

Figure 1.11 – A Payroll Program

```
1. FILE IS PAYROLL
2. LIST BY DEPARTAMENT NAME ID SEX NET GROSS
3. SELECT DEPARTMENT = 911,914
4. SUBTOTALS BY DEPARTMENT
5. TITLE: “PAYROLL FOR DEPARTMENTS 911,914”
6. COLUMN HEADINGS: “DEPARTMENT”, “EMPLOYEE, NAME”;
   “EMPLOYEE, NUMBER”; “SEX”; “NET, PAY”; “GROSS, PAY”
```

Figure 1.12 – Query-Language Program to Produce Report of Figure 1.11. Each instruction is discussed in detail in the text

♦ Instruction 1 specifies that the payroll data are stored on a FILE called PAYROLL. Although the data of only one file are needed in this example, requests requiring data from several files are no more difficult.

♦ Instruction 2 specifies that the information in the report is to be sorted (department 911 before 914) and LISTed BY DEPARTMENT. It also specifies which data elements within the file are to be included in the report of Figure 1.11. If the instruction had been LIST BY DEPARTMENT BY NAME, then the employee names would be listed in alphabetical order for each department.

♦ Instruction 3 specifies the criterion by which records are SELECTed. The personnel manager is interested only in those employees from DEPARTMENTs 911 and 914. Other criteria could be included for further record selection. For example, the criterion "GROSS>400.00" could be added to select only those people (from departments 911 and 914) whose gross pay is greater than $400.00.

♦ Instruction 4 causes SUBTOTALS to be computed and displayed BY DEPARTMENT.

♦ Instructions 5 and 6 allow the personnel manager to improve the appearance and readability of the report by including a title and labelling the columns. Instruction 5 produces the report by including a title and labelling the columns. Instruction 5 produces the report title, and instruction 6 specifies descriptive column headings.

The COBOL equivalent of this request would require over 150 lines of code!

Query languages are effective tools for generating responses to a variety of request for information. Short query-language programs, similar to the one in Figure 1.12, are all that is needed to respond to the following typical management requests:

♦ Which employees have accumulated over 20 sick days since January 1?

♦ Are there any deluxe single hospital rooms to be vacated by the end of the day?

♦ What is a particular student's average in all English courses taken?

♦ List departments that have exceeded their budget alphabetically by the department head's name.

Entrepreneurial Innovation

Procedure-oriented languages, such as FORTRAN and COBOL, were designed by volunteer committees and individuals, primarily for the public domain. Companies, such as CDC and IBM, then developed compilers and interpreters to support these languages. The fourth-, fifth-, and sixth-generation languages are products of entrepreneurial innovation. That is, these languages (e.g., NOMAD2, EASYTRIEVE Plus) and developed to be marketed and sold. The demand for very high-level languages is so great that many entrepreneurs have produced products for this highly competitive market. Each of the last three produced products for this competitive market. Each of the last three generations may have a dozen or more equally popular languages. Customers will, or course, purchase a language that will best meet their information processing needs.

1.7. The Fifth Generation: Application Generators

Principles and Use. Application generators are designed primary for use by computer professionals. The concept of an application generator is not well defined, nor will it ever be, as entrepreneurs are continually working to provide better ways to create information systems. In contrast to the ad-hoc orientation (one-time information requests) of fourth-generation query languages, application generators are designed to assist in the development of full-scale information systems.

During the development of an information system with an application generator, also called a code generator, programmers specify, through an interactive dialogue with the system, what information processing tasks are to be performed. This is essentially a fill-in-blank process. After programmers enter their specification, the actual procedure-level instructions are automatically generated. In the creation of an information system, you would describe the data base, then specify screen layouts for file creation and maintenance, data entry, management reports, and menus. The application generator software consist of modules of reusable code that are pulled together and integrated automatically to complete the system.

Improved Productivity. Application generators are currently in the infant stage of development. Existing application generators do not have the flexibility of procedure-oriented languages; therefore, the generic reusable code of application generators must occasionally be supplemented

with custom code to handle unique situations. Normally, about 10% to 15% of the code would be custom code. Application generators provide the framework by which to integrate custom code with generated code.

When used for the purposes intended, application generators can increase programmer and system analyst productivity by as much as 500%. As they mature, application generators will play an ever-increasing role in information systems development.

1.8. The Sixth Generation: Natural Languages

The next step in the sophistication of programming languages is the natural language: the sixth generation. The premise behind a natural language is that the programmer or user needs little or no training. He or she simply writes, or perhaps verbalizes, specifications without regard for instruction format or syntax. To date, there is no such language. Researches are currently working to develop pure natural languages that will permit an unrestricted dialog between us and a computer. Although the creation of such a language is difficult to comprehend, it is probably inevitable.

In the meantime, natural languages with certain syntax restrictions are available. And for limited information processing tasks, such as ad-hoc inquiries and report generation for a specific application area, existing natural languages work quiet well. With certain limitations, existing natural languages permit users to express queries in normal, everyday English. You can phrase a query any way you want. For example, you could say, “Let me see the average salaries by job category in the marketing department”. Or you would get the same results if you said, “What is the average salary in the marketing department for each job classification?” If your query is unclear, the natural-language software might ask you questions that will clarify any ambiguities.

A natural language interprets many common words, but other words peculiar to a specific application or company would have to be added by the user. All common and user-supplied words comprise the lexicon, or the dictionary of words that can be interpreted by the natural language. The sophistication of types of queries that can be accepted are above inquiry, the words “Let”, “me”, and “see”, their meaning, and the context in which they are used would have to be entered into the lexicon before the phrase “Let me see” could be interpreted by the natural-language software. Also,

in the above example, “category” and “classification” must be defined in the lexicon to mean the same thing.

A natural-language equivalent of the query-language program of Figure 1.12 would be: “Show me a report of employee payroll data for departments 911 and 914”. Department and overall summary data are automatically generated. Usually, state-of-the-art natural-language software can interpret no more than a one-sentence query at a time. Other typical natural-language queries might be:

- Are there any managers between the ages of 30 and 40 in the north-west region with MBA degrees?
- Show me a pie chart that compares voter registrations for the southern states.
- What are the top ten best-selling fiction books in California?

1.9. The Operating System: the Boss

Just as the processor is the nucleus of the computer system, the operating system is the nucleus for all software activity. The operating system is a family of system software programs that are usually, though not always, supplied by the computer system vendor.

Mainframe Operating System

Design Objectives. All hardware and software, both system and applications, are under the control of the operating system. You might even call the operating system the “boss”. The logic, structure, and nomenclature of the different operating system vary considerably. However, each is designed with the same three objectives in mind.

1. Minimize turnaround time [elapsed time between submittal of a job (e. g., print payroll checks) and receipt of output].
2. Maximize throughput (amount of processing per unit time).
3. Optimize the use of the computer system resources (processor, primary storage, and peripheral devices).

The Supervisor. One of the operating system programs is always resident in primary storage (see Figure 1.13). This program, called the supervisor, loads other operating system and applications programs to primary storage as they are needed. For example, when you request a

COBOL program compilation, the supervisor loads the COBOL compiler to primary storage and links your source program to the compiler to create an object program. In preparation for execution, another program, the linkage editor, assigns a primary storage to each byte of the object program.

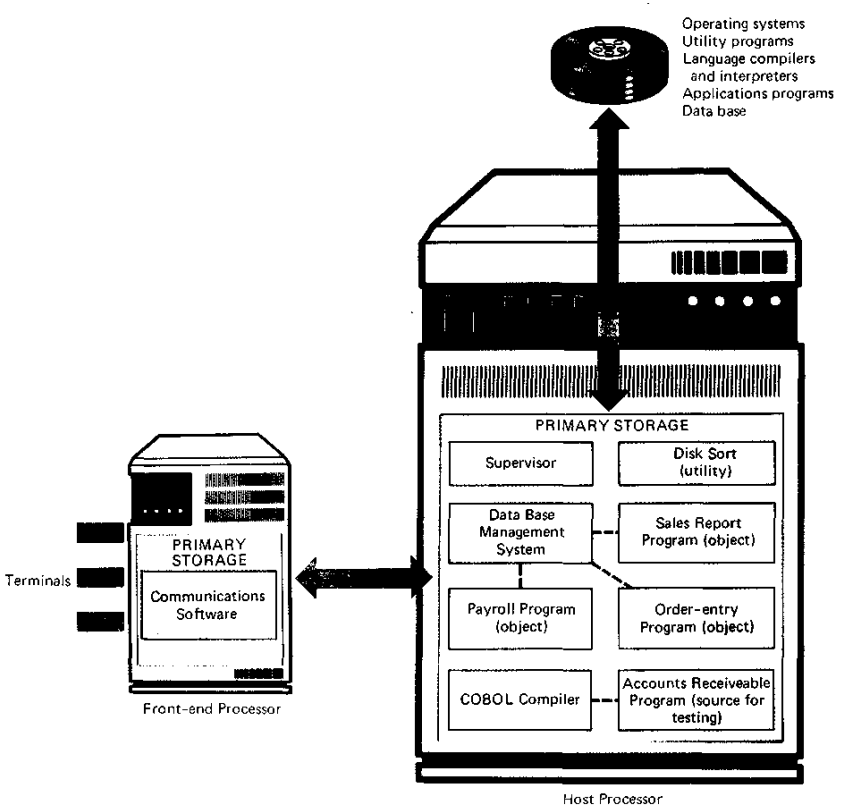


Figure 1.13 – Software, Storage, and Execution. The supervisor program is always resident in primary storage and calls other programs, as needed, from secondary storage. All programs must be in object format to be executed. Applications programs rely on data base management system software to assist in the retrieval of data from secondary storage. Software in the front-end processor handles data communications-related tasks

Allocating Computer Resources. In a typical computer system, several jobs will be executing at the same time. The operating system determines which computer system resources are allocated to which

programs. As an example, suppose that a computer system with only one printer has three jobs whose output is ready to be printed. Obviously two must wait. The operating system continuously resolves this type of resource conflict to optimize the allocation of computer resources.

Operator Interaction. The operating system is in continuous interaction with computer operators. The incredible speed of a computer system dictates that resource-allocation decisions be made at computer speeds. Most of these decisions are made automatically by the operating system. For decisions requiring human input, the operating system interrogates the operators through the operator console. The operating system also sends messages to the operator. A common message is "Printer no. 1 is out of paper".

Compatibility Considerations. There are usually several operating system alternatives available for medium and large computers. The choice of an operating system depends on the processing orientation of the company. Some operating systems are better for timesharing, others for batch processing, and other for distributed processing.

Applications programs are not as portable between operating systems as we would like. An information system is designed and coded for a specific compiler, computer, and operating system. This is true for both micros and mainframes. Therefore, programs that work well under one operating system may not be compatible with a different operating system.

Microcomputer Operating Systems

The objectives and functions of microcomputer operating system are similar to mainframe operating systems. Although some vendors of microcomputers supply their own operating systems, many use MS DOS, CP/M, or UNIX. MS DOS, developed by Microsoft Corporation, CP/M, developed by AT&T, have become the unwritten standards for the microcomputer industry. MS DOS and CP/M have a long tradition of acceptance in the single-user microcomputer environment. UNIX has a similar tradition of acceptance in the multiuser mainframe environment.

Some of today's micros are more powerful than the UNIX-based mainframe computers of the early 1980s. This increased capacity enabled micro vendors and AT&T to adapt the very popular UNIX operating system for multiuser, multitasking (able to handle different processing tasks at the same time) microcomputers. Of course, MS DOS and CP/M have been upgraded to handle the multiuser, multitasking environment also. You

may encounter spinoffs of these operating system. For example, PC DOS for the IBM PC is based on Microsoft's MS DOS, and XENIX is spinoff of UNIX.

Because these operating systems are so widely used, hundreds of software vendors have developed systems and application software that are compatible with them.

Before you can use a minicomputer, you must “boot the system”. The procedure for booting the system on most micros is to simply load the operating system from the disk into primary storage, then flip the ‘on” switch. A few seconds later, with the operating system in memory, you are ready to begin processing.

Other System Software Categories

In this chapter we have discussed two system software categories: programming-language compiler/interpreters and operating systems. In this section we discuss three more: utility programs, communications software, and data base management system software.

Utility Programs. Utility programs are service routines that make life easier for us. They eliminate the need for us to write a program every time we need to perform certain computer operations. For example, with the help of utility programs, an employee master file can be easily "dumped" or copied from magnetic disk to magnetic tape for backup, or the employee master file can be sorted by social security number.

Communications Software. Communications software controls the flow of traffic (data) to and from remote locations. In small computers this is an operating system function, but in mainframe computer system the communications software is usually separate (see Figure 1.3). Communications software is executed on the front-end processor, the down-line processor, and the host processor. Functions performed by communications software include: preparing data for transmission (i.e., inserting start/stop bits in messages), polling remote terminals for input, establishing the connection between two terminals, encoding and decoding data, and parity checking.

Data Base Management Systems (DBMS). Data base management system (DBMS) software provides the interface between application programs and the data base. If you want to write a program to update employee records, the only instructions that you will need to retrieve a record are those required to accept the employee’s name at a workstation.

Once the employee's name is entered, the data base management system software does the rest. The employee record is retrieved from secondary storage and moved to primary storage for processing. Control is then returned to your application program to complete the update.

In effect, a programmer does not have to be concerned with finding or retrieving data. This is handled by the data base management system software and is transparent to the programmer; that is, it is done automatically without programmer intervention.

1.10. Software Concepts

Multiprogramming

All computers, except small micros, have multiprogramming capability. Multiprogramming is the seemingly simultaneous execution of more than one program at a time. We have learned that a computer can execute only one program at a time. But the internal processing speed of a computer is so fast that several programs can be allocated a "slice" of computer time in rotation; this makes it appear to us that several programs are being executed at once.

The great difference in processor speed and the speeds of the peripheral devices makes multiprogramming possible. A 40,000-line-per-minute printer cannot even challenge the speed of an average mainframe processor. The processor is continually waiting for the peripheral devices to complete such tasks as retrieving a record from disk storage, printing a report, or copying a backup file onto magnetic tape. During these "wait" periods, the processor just continues processing other programs. In this way, computer system resources are used efficiently.

In a multiprogramming environment, it is not unusual for several programs to require the same I/O device. For example, two or more programs may be competing for the printer. Rather than hold up the processing of a program by waiting for the printer to become available, both programs are executed and the printer output for one is temporarily loaded to magnetic disk. As the printer becomes available, the output is called from magnetic disk and printed. The process is called spooling.

Virtual Memory

We learned in Chapter, "Inside the Computer", that all data and programs must be resident in primary storage to be processed. Therefore,

primary storage is a critical factor in determining the throughput, or how much work can be done by a computer system per unit of time. Once primary storage becomes full, no more programs can be executed until a portion of primary storage is made available.

Virtual memory is a system software addition to the operating system that effectively expands the capacity of primary storage through the use of software and secondary storage. This allows more data and programs to be resident in primary storage at any given time.

The principle behind virtual memory is quite simple. Remember, a program is executed sequentially-one instruction after another. Programs are segmented into pages, so only that portion of the program being executed is resident in primary storage. The rest of the program is on disk storage. Once the instructions have been executed in the page that is resident in primary storage, or control is passed to an instruction in another page, the appropriate page is rolled (moved) into primary storage from disk storage. This page replaces the previous page, and program execution continues.

The advantage of virtual memory is that primary storage is effectively enlarged, giving programmers greater flexibility in what they can do. For example, some applications require several large programs to be resident in primary storage at the same time (for example, order-processing and credit-checking programs). If the size of these programs exceeds the capacity of “real” primary storage, then virtual memory can be used as a supplement to complete the processing.

The disadvantage of virtual memory is the cost in efficiency during program execution. A program that has many branches to many pages will execute slowly because of the time required to roll pages from secondary to primary storage. Excessive page movement results in too much of the computer's time being devoted to page handling and not enough to processing. This excessive data movement is appropriately named thrashing and can actually be counterproductive.

Task II. Fill in the spaces with proper words from the box.

1. A _____ directs a computer to perform certain operations.
2. The program is produced by a _____ who uses any of variety of _____ to communicate with the computer.
3. Application generators consist of modules of _____ that are pulled together and integrated automatically to complete the system.

4. A Cobol program has _____ (how many) divisions.

5. An _____ performs a function similar to a compiler but it translates one instruction at a time.

6. The features of_____ include Englishlike instructions, limited mathematical manipulation of data, automatic report formatting, sequencing and record selection by criteria.

7. _____ languages are more emphasized on input and desired output than on the procedures involved.

8. All hardware and systems and applications software are under control of the _____.

9. Operating systems are oriented to a particular type of _____, such as timesharing, batch, or distributed processing.

10. A _____, will someday enable programmers to write or verbalize program specifications without regard to instruction format or syntax.

(natural language, processing environment, programmer, reusable code, query language, operating system, programming languages, interpreter, problem-oriented, program, four)

Task III. Decide whether the following statements are true or false. If you think a statement is false, change it to make it true.

1. Systems software is designed to perform specific personal, business, or scientific processing tasks.

2. Wren programming in one of the first three generations of languages you tell the computer “what to do”, not “how to do it”.

3. Assembler-level languages are instructions coded as a series of 1s and 0s.

4. An object program is always free of logic errors.

5. A fourth generation program will normally have fewer instructions than the same program written in a third-generation language.

6. Application generators are used almost exclusively for ad-hoc requests for information.

7. An individual must undergo extensive training before he or she can write programs in a natural language.

8. The operating system program that is always resident in main memory is called the supervisor.

9. Programs are segmented into pages before they are spooled.

10. The procedure for booting the system on most microcomputers is not needed.

Task IV. Answer these questions to the text.

1. Associate each of the following with a particular generation of languages: lexicon, mnemonics and Ada.
2. What are the four divisions in a Cobol program? Which one contains the program logic?
3. Program diagnostics identify what two types of program errors?
4. Name two procedure-oriented programming languages in each of the three classification areas-business, scientific, and multipurpose.
5. What are the programs called that translate source programs to machine language? Which one does the translation on a single pass? Which one does it one statement at a time?
6. What term is used to refer to the programs that direct the activities of the computer system?
7. Which programming language must be entered on specially designed terminals? Why?
8. Contrast fourth-generation to fifth-generation languages.
9. What are the design objectives of an operating system?
10. Why is it necessary to spool output in a multiprogramming environment?
11. Give two examples each of applications and systems software.
12. What are applications generators designed for?
13. Name the systems software category associated with: (a) the organization's data base, (b) file backup, and (c) overall software and hardware control.

Task V. Discuss the following matters.

1. The difference between a program and a programming language.
2. System software categories such as utility programs, communications software and data base management system software.
3. If each new generation of languages enhances interaction between programmers and the computer, why not write programs using the most recent generation of languages?

4. Which generations of languages would a public relations manager be most likely to use? Why?

5. Suppose you are a programming manager and find that 12 of the 16 programmers would prefer to switch to Cobol from RPG. Would you support the switch, given that all 600 existing programs are written in RPG and only three programmers are proficient in Cobol? Why or why not?

6. The principle behind virtual memory. Advantage and disadvantage of virtual memory.

7. Advantage of interpreters over compilers and their disadvantage.

UNIT 2. PROGRAMMING CONCEPTS

Task I. Before reading the text translate the words and word combinations and match them with the appropriate explanation.

1. Structured programming	– a) subordinate module.
2. Flowchart	– b) permits data to be assigned, i.e. transfer red to a named primary storage location.
3. Driver module	– c) programs are developed for the team and the project and his or her ego.
4. Subroutine	– d) syntax or logic error in program.
5. Sequence structure	– e) logic of the program is addressed hierarchically in logical modules and then logic of each module is translated into a sequence of program instructions that can be executed independently.
6. Selection structure	– f) causes control of execution to “go to” another portion of the program.
7. Loop structure	– g) represents the logic in programlike statements written in plain English.
8. Pseudocode	– h) GOTO statements are not permitted to use.

9. Decision table	–	i) sequence of instructions that are executed one after another.
10. Egoless programming	–	j) causes other program modules to be executed as they are needed.
11. Statements	–	k) riding a program of bugs.
12. Unconditional instruction	–	l) the processing steps are performed in sequence.
13. Conditional instruction	–	m) used to represent the program logic when a portion of the program is to be executed repeatedly until a particular condition is met.
14. GOTO statement	–	n) if certain conditions are met, then a branch is made to a certain part of the program. They are referred to as IF statements.
15. GOTO-less programming	–	o) illustrates data, information and work flow through the interconnection of specialized symbols with flow lines.
16. Assignment instruction	–	p) points you in the right direction, but some extra effort may be required to isolate the exact cause of the error.
17. Bug	–	q) a tool used to depict what happens in a system or program for occurrences of various circumstances. It is based on “IF … THEN” logic.
18. Debugging	–	r) disrupts the normal sequence of execution by causing an unconditional branch to another part of the program or to a subroutine.
19 Error message	–	s) depicts the logic for selecting the appropriate sequence of statements.

2.1. Programming in perspective

A computer is not capable of performing calculations or manipulating data without exact step-by-step instructions. These instructions take the form of a computer program. Five, fifty, or even several hundred programs may be required for an information system. Electronic spreadsheet software is made up of dozens of programs that work together so that you can perform spreadsheet tasks. The same is true of word processing software.

Most of the programs that you develop while you are a student will be independent of those developed by your classmates and, more often than not, independent of one another. In a business environment, programs are often complementary to one another. For example, you might write a program to collect the data and another program to analyze the data and print a report.

There is no such thing as an “easy” program. A programming task, whether it be in the classroom, in business, or at home, should challenge your intellect and logic capabilities. As soon as you develop competence at one level, your instructor will surely assign you a program that is more difficult than anything you have done in the past. Even when doing recreational programming on your personal computer, you won’t be satisfied with an “easy” program. You will probably challenge yourself with increasingly complex programs.

Programming can be enormously frustrating, especially at first. Don’t despair! Just when you think that the task confronting you is impossible, a little light will turn on and open the door to the joy of learning to program. That light has turned millions of people on to programming, and it will turn you on, too (if it hasn’t already).

2.2. Problem Solving and Programming Logic

A single program addresses a particular problem: to define dance movements, to compute and assign grades, to permit an update of a data base, to monitor a patient's heart rate, to analyze marketing data, and so on. In effect, when you write a program, you are solving a problem. To solve the problem you must derive a solution. And to do that, you must use your powers of logic.

A program is like the materials used to construct a building. Much of the brainwork involved in the construction of a building goes into the blueprint. The location, appearance, and function of a building are determined long before the first brick is laid. And so it is with programming. The design of a program, or its programming logic (the blueprint), is completed before the program is written (or the building is constructed). This section and the next discuss approaches to designing the logic for a programming task. Later in this chapter we'll discuss the program and the different types of program instructions.

Structured Program Design: Divide and Conquer

Figure 2.1 illustrates a structure chart for a program to print weekly payroll checks. Hourly and commission employees are processed weekly. A structure chart for a program to print monthly payroll checks for salaried employees would look similar, except that task 1.2, compute gross earnings, would not be required. The salary amount can be retrieved directly from the employee database.

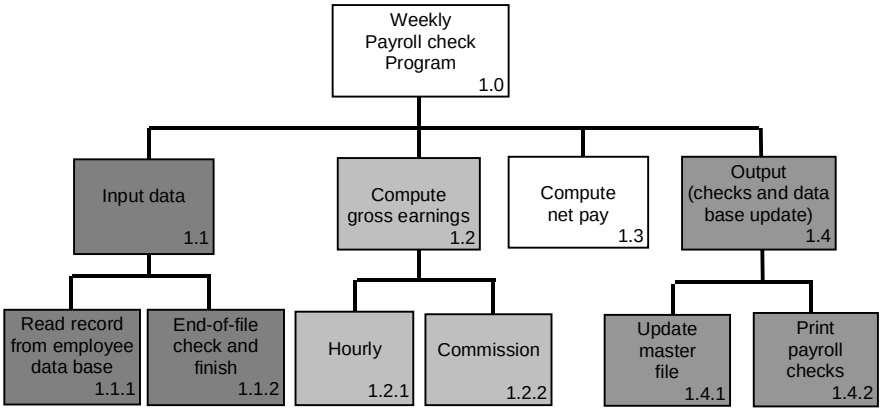


Figure 2.1 – Program Structure Chart. The logic of a payroll program to print weekly payroll checks can be broken into modules for ease of understanding, coding, and maintenance

The structure chart permits a programming problem to be broken down into a hierarchy of tasks. A task can be broken down into subtasks, as long as a finer level of detail is desired. The most effective programs are designed so they can be written in modules, or independent tasks. It is much easier to address a complex programming problem in small, more

manageable modules than as one big task. This is done using the principles of structured programming.

In structured programming, the logic of the program is addressed hierarchically in logical modules (see Figure 2.1). In the end, the logic of each module is translated into a sequence of program instructions that can be executed independently. By dividing the program into modules, the structured approach to programming reduces the complexity of the programming task. Some programs are so complex that, if taken as a single task, they would be almost impossible to conceptualize, design, and code. Again, we must “divide and conquer”.

2.3. Program Design Techniques

A number of techniques are available to help programmers analyze a problem and design the program. HIPO and data flow diagrams can be used as program design aids as well as system design aids. Both can graphically capture the logic of systems (the general level) or programs (the detailed level). The process symbol in a system data flow diagram might represent one or several programs (e.g., print payroll checks), whereas a process symbol in a program data flow diagram might represent a computation (e.g., compute federal tax deduction). In this section, such design techniques as flowcharting, pseudocode, and decision tables-are presented as they would be used in the design of a program's logic. These techniques can also be used as system design tools.

Flowcharting

One of the most popular design techniques is flowcharting. Flowcharts illustrate data, information, and work flow through the interconnection of specialized symbols with flow lines. The combination of symbols and flow lines portray the logic of the program or system. The more commonly used flowchart symbols are shown in Figure 2.2.

Flowcharting Symbols. Each symbol indicates the type of operation to be performed, and the flowchart graphically illustrates the sequence in which the operations are to be performed. Flow lines → depict the sequential flow of the program logic. A rectangle ▭ signifies some type of computer process. The process could be as specific as “compute an individual’s grade average” (in a program flowchart) or as general as “prepare class schedules for the fall semester” (in a system flowchart).

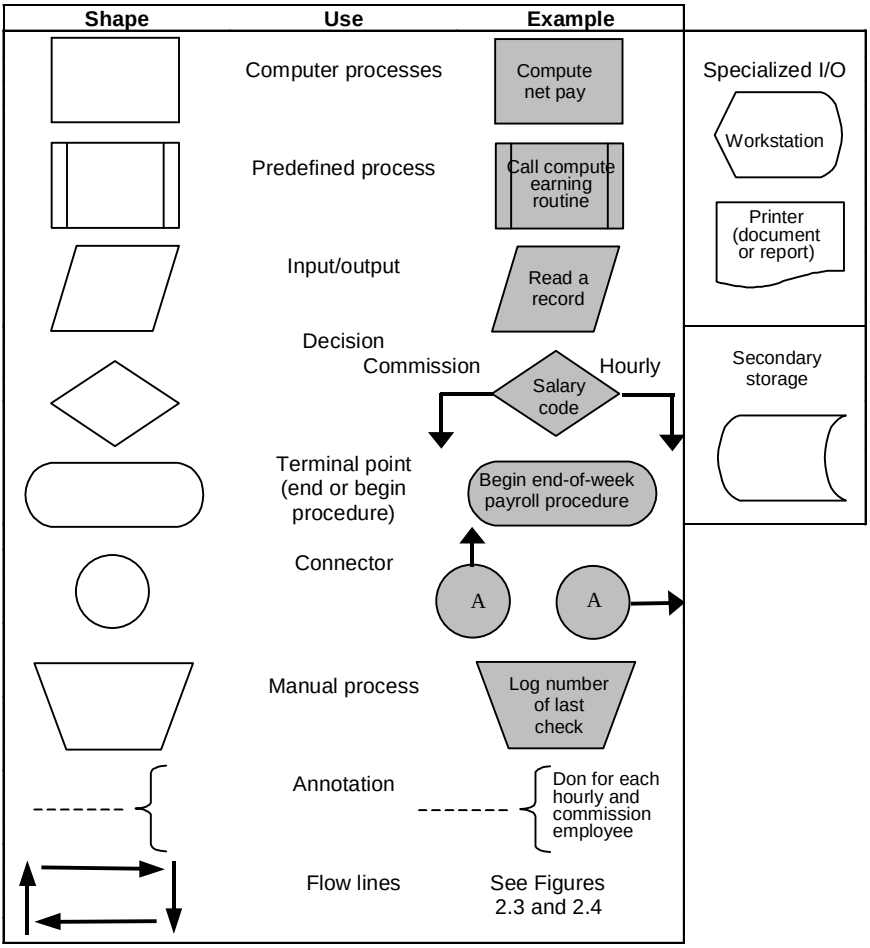


Figure 2.2 – Flowchart Symbols

The predefined process  , a special case of the process symbol, is represented by a rectangle with extra vertical lines. The predefined process refers to a group of operations that may be detailed in a separate flowchart. The parallelogram  is a generalized input/ output symbol that denotes any type of input to, or output from, the program or system. The diamond-shaped symbol  marks the point at which a decision is to be made. In a program flowchart, a particular set of instructions is executed based on the outcome of a decision. For example, in a payroll program, gross pay is

computed differently for hourly and commission employees; therefore, for each employee processed, a decision is made as to which set of instructions are to be executed.

Each flowchart must begin and end with the oval terminal point symbol  . A small  circle is a connector and is used to break and then link flow lines. The connector symbol is often used to avoid having to cross lines. The trapezoid  indicates that a manual process is to be performed. Contrast this to a computer process represented by a rectangle. The bracket ---  permits descriptive notations to be added to flowcharts.

The on-line data storage symbol  represents a file or data base. The most common specialized input/output symbols are the workstation  and the printer (hard copy)  symbols.

These symbols are equally applicable to system and program flowcharting and can be used to develop and represent the logic for each. A system flowchart for a payroll system is illustrated in Figure 2.3. Contrast this system flowchart to the program flowchart of Figure 2.4. The program flowchart portrays the logic for the structure chart of Figure 2.1. The company in the example of Figure 2.1 processes hourly and commission employee checks each week (salary employee checks are processed monthly). Gross earnings for hourly employees are computed by multiplying hours-worked times the rate of pay. For salespeople on commission, gross earnings are computed as a percentage of sales.

The driver module in structured programming, each program has a driver module that causes other program modules to be executed, as they are needed. The driver module for our example payroll program (see Figure 2.4) is a loop that “calls” each of the subordinate modules, or subroutines, as they are needed for the processing of each employee. The program is designed such that when the payroll program is initiated, the “input data” module (1.1) is executed, or “performed” first. After execution, control is then returned to the driver module, unless there are no more employees to be processed, in which case execution is terminated (the “Finish” terminal point). For each hourly or commission employee, Modules 1.2, 1.3, and 1.4 are performed, and at the completion of each subroutine, control is passed back to the driver module.

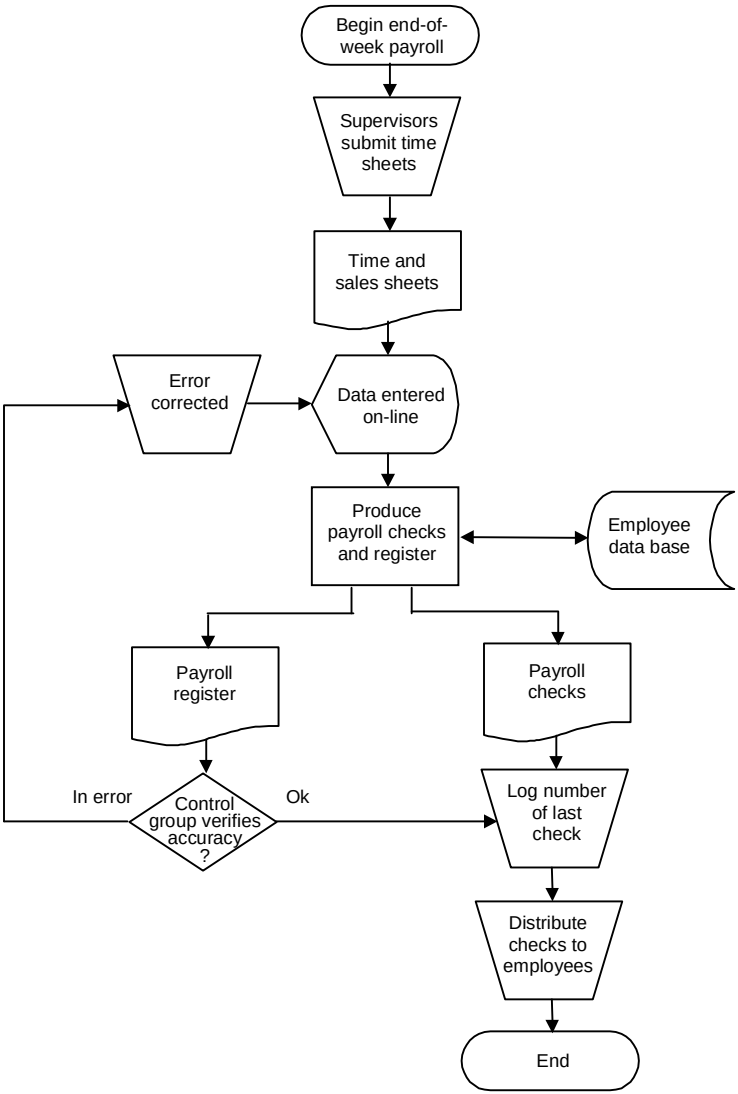


Figure 2.3 – General Systems Flowchart. This flowchart graphically illustrates the relationship between I/O and major processing activities in a payroll system

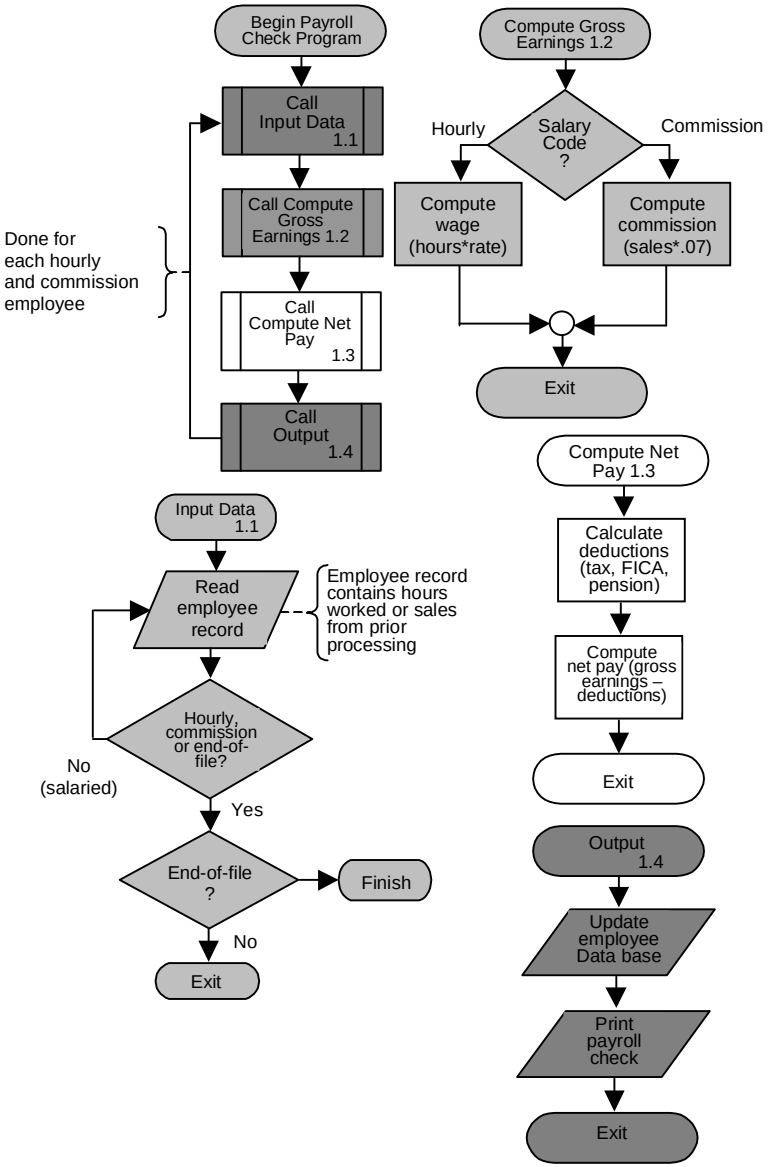


Figure 2.4 – Program Flowchart. This flowchart portrays the logic of a program to compute and print payroll checks for commission, hourly, and salaried employees (see the structure chart of Figure 2.1). The logic is designed so that a driver module calls subroutines as they are needed to process each employee

When dividing the logic of a program into modules, a good guideline to follow is that each module should have a single entry point and a single exit point. That is, a program module should begin and end with the same instructions each time it is executed. Thus, transfer of control into and out of a particular program module occurs only at the entry and exit points. The single-entry/single-exit-point guideline encourages good program logic, because it does not permit multiple branches to other modules. An excess of branches tends to confuse the logic of the program. The modules that begin or end a program may have an extra entry/exit point to begin or end program execution.

Programming Control Structures. Through the 1970s, programmers unknowingly wrote what is now referred to as "spaghetti code". It was so named because their program flowcharts appeared more like a plate of spaghetti than a logical analysis of a programming problem. The redundant and unnecessary branching (jumps from one portion of the program to another) of a spaghetti-style program resulted in confusing logic, even to the person who wrote it. These programs were difficult to write and debug, and even harder to maintain.

Computer scientists thwarted this dead-end approach to developing program logic by identifying three basic control structures into which any program, or subroutine, can be segmented. By conceptualizing the logic of a program in these three structures-sequence, selection, and loop-programmers can avoid writing spaghetti code and thus produce programs that can be more easily understood and maintained. The use of these three basic control structures has paved the way for a more rigorous and scientific approach to solving a programming problem. These three control structures are illustrated in Figures 2.5, 2.6, and 2.7, and their use is demonstrated in the payroll example of Figure 2.4.

Sequence Structure. In the sequence structure (Figure 2.5), the processing steps are performed in sequence, one after another. Modules 1.3 and 1.4 in Figure 2.4 are good examples of sequence structures.

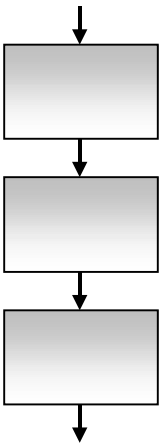


Figure 2.5 – Sequence Control Structures

Selection Structure. The selection structure (Figure 2.6) depicts the logic for selecting the appropriate sequence of statements. In Figure 2.4, our example payroll program, the selection structure is used to illustrate the logic for the computation of gross pay for hourly and commission employees (Module 1.2). In the selection structure, a decision is made as to which sequence of instructions is to be executed next. In Module 1.2, is the salary code for an employee record "hourly" or "commission"?

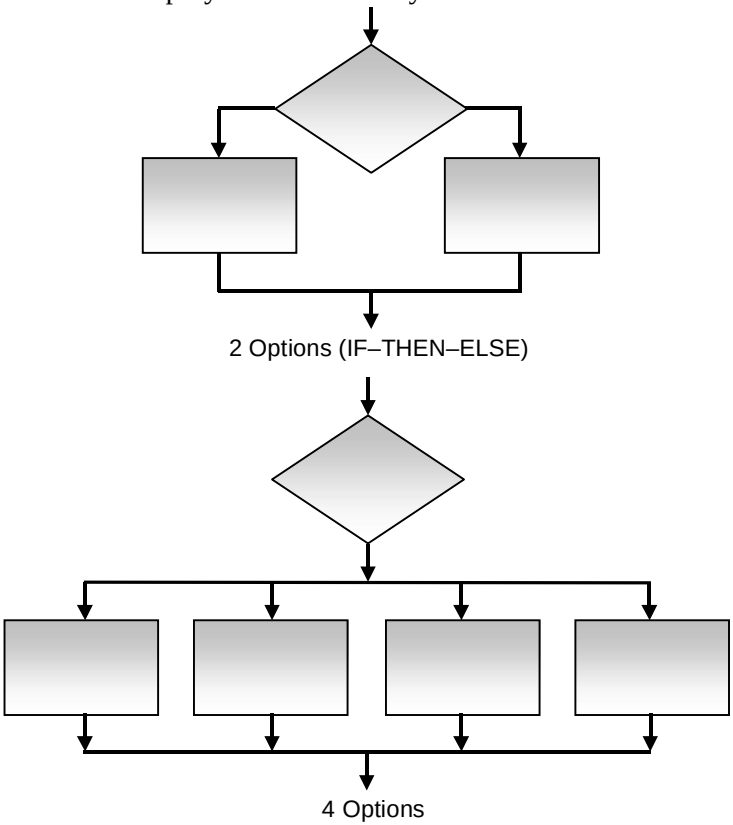


Figure 2.6 – Selection Control Structures. Any number of options can result from a decision in a selection control structure

In the actual payroll program, a different sequence of instructions is executed for hourly and commission employees. The two-option selection structure is sometimes referred to as IF-THEN-ELSE. For example, IF the

salary code is hourly, THEN compute wages, ELSE compute commission (employee pay type is commission by default).

The selection structure of Module 1.2 presents two decision options: hourly or commission. Other circumstances might call for three or more decision options. For example, suppose part-time hourly employees are treated differently than full-time hourly employees, and salary employees are also paid weekly. In this case, there would be four decision options: hourly (full-time), hourly (part-time), commission, and salary.

Loop Structure. The loop structure (Figure 2.7) is used to represent the program logic when a portion of the program is to be executed repeatedly until a particular condition is met. There are two variations of the loop structure (see Figure 2.7): when the decision, or test-on-condition, is placed at the beginning of the statement sequence, it becomes a DOWHILE loop; when placed at the end, it becomes a DOUNTIL loop (pronounced do while and do until). Notice that the leading statements in a DOUNTIL structure will always be executed at least once. In the example payroll flowchart of Figure 2.4, that portion of the input data module (1.1) that reads an employee record is illustrated in a DOUNTIL loop. Employee records, containing hours-worked and sales data, are read sequentially. Since only hourly or commission employees are processed weekly, the loop is repeated until the record of an hourly or commission employee is read, or until the end-of-file marker is reached. When an hourly or

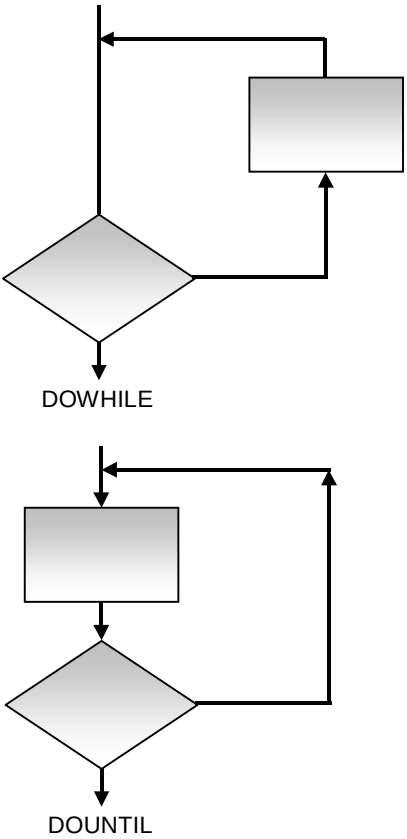


Figure 2.7 – Loop Control Structures. The two types of loop structures are DOWHILE and DOUNTIL

commission employee record is read, control is returned to the driver module, which in turn passes control to Module 1.2.

Level of Flowchart Detail. The example program flowchart of Figure 2.4 is made somewhat general so that the concepts can be demonstrated more easily. A flowchart showing greater detail could be compiled, if desired. For example, Figure 2.8 illustrates how Module 1.3, Compute Net Pay, can be expanded to show more detail.

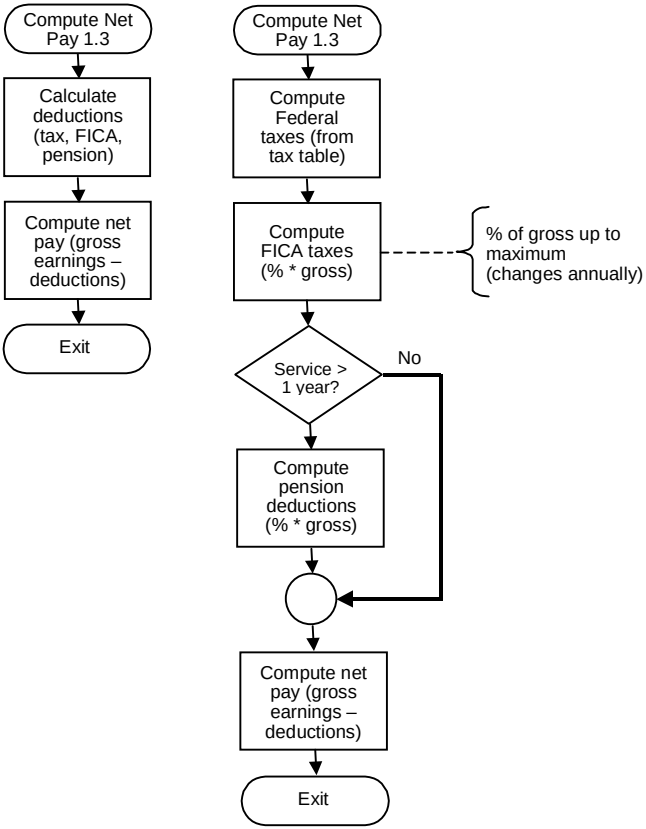


Figure 2.8 – Level of Detail in Flowcharts. The logic of Module 1.3, Compute Net Pay, of Figures 2.1 and 1.4 is depicted in a general and a more detailed flowchart

In program flowcharting, the level of detail is a matter of personal preference. Some programmers complete a general flowchart that outlines the overall program logic, then flesh it out with a more detailed flowchart.

Pseudocode

Another design technique that is used almost exclusively for program design is called pseudocode. While the other techniques represent the logic of the program graphically, pseudocode represents the logic in programlike statements written in plain English. Since pseudocode does not have any syntax guidelines (i.e., rules for formulating instructions), you can concentrate on developing the logic of your program. Once you feel that the logic is sound, the pseudocode is easily translated to a procedure-oriented language that can be executed. In Figure 2.9, the logic of a simple program is represented in pseudocode and with a flowchart.

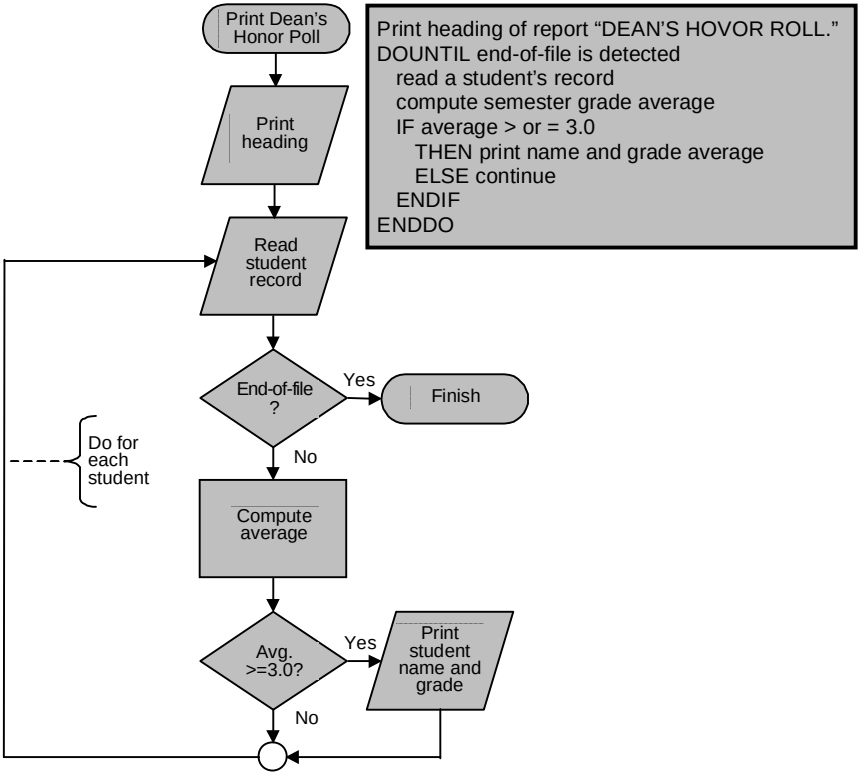


Figure 2.9 – Pseudocode Example with Flowchart. This pseudocode program depicts the logic of a program to compile a list of students who have qualified for the dean's honor roll. The same logic is show in a flowchart

Decision Tables

The decision table is a handy tool that analysts and programmers use to graphically depict what happens in a system or program for occurrences of various circumstances. The decision table is based on "IF … THEN" logic. IF this set of conditions is met, THEN take this action. Decision tables are divided into quadrants (see Figure 2.10). Conditions that may occur are listed in the condition stub (the upper left quadrant). The possible occurrences for each condition type are noted in the condition entries (the upper right quadrant). Each possible set of conditions, called a rule, is numbered at the top of each column. Actions that can result from various combinations of conditions, or rules, are listed in the action stub (the lower left quadrant).

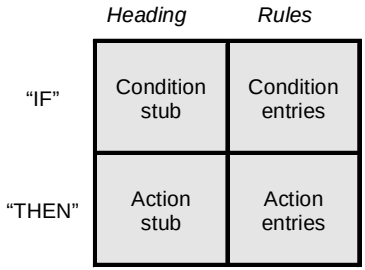


Figure 2.10 – Decision-Table Format. Decision tables are divided into four quadrants: condition stub, condition entries, action stub, and action entries

For each rule, an action-to-be-taken entry is made in the action entries (the lower right quadrant).

The decision table in Figure 2.11 illustrates what action is taken for each of several sets of conditions. For example, IF the employees to be processed are salaried and it is the end of the month (rule 1), THEN both paychecks and a payroll register are printed. IF the employees to be processed are on a commission and it is the end of the week (rule 4), THEN paychecks only are printed.

The decision table is not a good technique for illustrating work flow. It is, however, very helpful when used in conjunction with flowcharts, data flow diagrams, and HIPO charts. The major advantage of decision tables is that a programmer or analyst must consider all alternatives, options, conditions, variables, and so on. With decision tables, the level of detail is dictated by the circumstances. With flowcharts and other design techniques,

the level of detail (contrast Figures 2.4 and 2.8) is more a matter of personal preference.

Payroll type/output chart	Rules				
	1	**2**	**3**	**4**	**5**
Salaried employee	Y	N	N	N	N
Hourly employee	N	Y	Y	N	N
Commission employee	N	N	N	Y	Y
End of week	N	Y	N	Y	N
End of month	Y	N	Y	N	Y
Print paychecks	X	X		X	X
Print payroll register	X	X	X		X

Figure 2.11 – Decision Table. This decision table depicts what payroll outputs would be generated for various payroll types and conditions

Egoless Programming

On occasion, a programmer might choose to write a program that will forever stand as a monument to how clever he or she can be. Unfortunately, “clever” programs detract from the effectiveness of a system. They may be clever to their author, but to the person who has to maintain the program, the logic may be indecipherable. To encourage teamwork and quality information systems, some companies have asked their programmers to adopt the principle of egoless programming. In short, this means that programs are developed for the team and the project, and not for the individual and his or her ego.

There is no substitute for good sound logic in programming. If you follow the guidelines of structured programming and make judicious use of program design techniques, your program will be easier to write, use of program design techniques, your program will be easier to write, use, and maintain.

Automated Design Tools

Software packages are now available that permit you to interactively create flowcharts, structure charts, and data flow diagrams on a display screen. For example, with data flow diagrams, you simply designate the type of symbol, where you want in to go, and its caption. The symbol and its caption are then displayed on the screen. You can create as many symbols as needed to depict the program or system logic. The symbols and flow lines can be moved, added, deleted, and revised as you test out various solutions to the problem.

Software packages are also available that automatically generate flowcharts directly from program code. The use of these generators is not recommended for original program development, but they may be useful when redocumenting programs whose documentation is out of date.

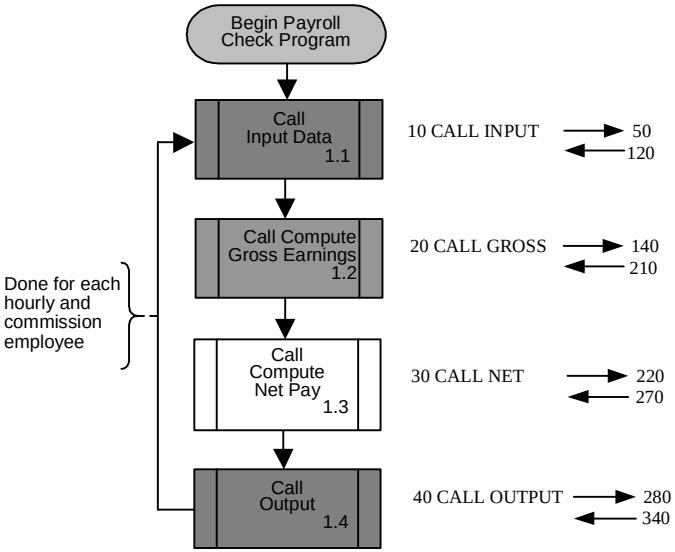
As for pseudocode, you probably already have access to all the software you need. Since pseudocode is simply text, any good word processing package will do nicely.

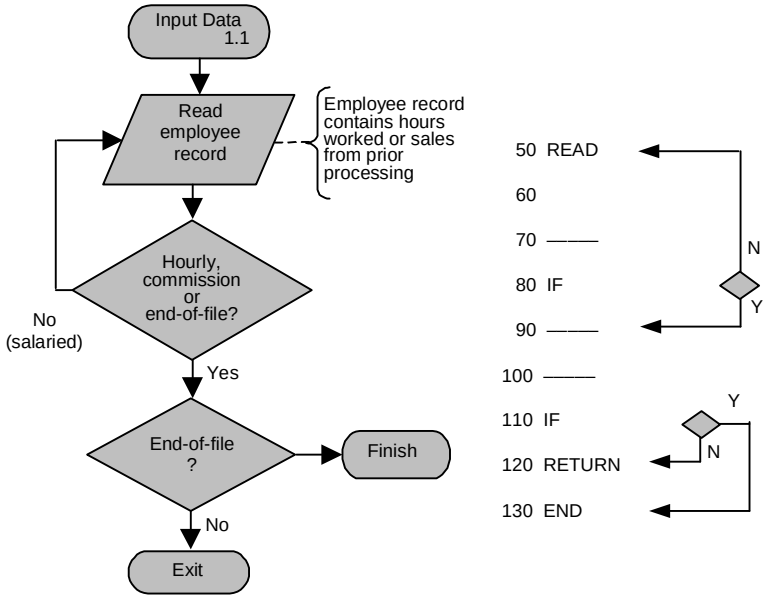
Such automated design tools are a welcome relief to the time-consuming task of manually documenting the logic of a program or system. And they sure save on erasures!

2.4. So what's a Program?: Concepts and Principles of Programming

A computer program is made up of a sequence of instructions that are executed one after another. These instructions, also called statements, are executed in sequence unless the order of execution is altered by a "test-on-condition" instruction or a "branch" instruction.

The flowchart of our example payroll program (Figure 2.4) is recreated in Figure 2.12, along with a sequence of language-independent instructions. Except for the computation of gross earnings, the processing steps are similar for both types of employees. Two sequences of instructions are needed to compute gross earnings for hourly and commission employees. We can also see from the flowchart that the sequence in which the instructions are executed may be altered at three places (decision symbols), depending on the results of the test-on-condition. In Module 1.2, for example, the sequence of instructions to be executed depends on whether the test-on-condition detects an hourly or commission employee.





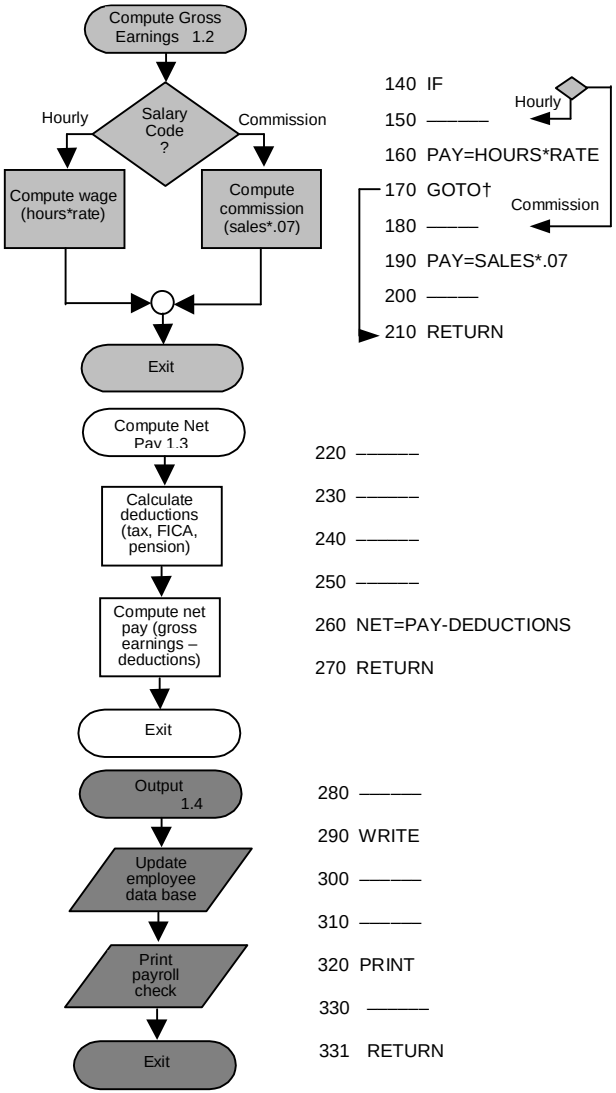


Figure 2.12 – Program Flowchart with Language-Independent Instructions. The flowchart (same as Figure 2.4) presents the logic of a payroll program to compute and print payroll checks for commission, hourly, and salaried employees. The accompanying "program" has a few language-independent instructions to help illustrate the concepts and principles of programming. A detailed discussion of this figure is presented in the text

To the right of the flowchart in Figure 2.12 is a representation of a sequence of language-independent instructions and the order in which they are executed. Statement numbers are included, as they are in most program listings. This program could be written in any procedure-oriented language. The purpose of the discussion below is to make you familiar with general types of programming instructions, and not those of any particular programming language. Each language has an instruction set with at least one instruction in each of the following instruction classifications: input/output, computation, control, data transfer and assignment, and format.

Input/Output. Input/output instructions direct the computer to "read from" or "write to" a peripheral device (e.g., printer, disk drive). Statement 50 of Figure 2.12 requests that an employee record, including pay data, be read from the data base. Statement 320 causes a payroll check to be printed.

Computation. Computation instructions perform arithmetic operations (add, subtract, multiply, divide, and raise a number to a power). Statement 160 (PAY = HOURS * RATE) computes gross earnings for hourly employees. Statement 190 (PAY = SALES * .07) computes gross earnings for commission employees, where the commission is 7% of sales.

Control (Decision and/or Branch). Control instructions can alter the sequence of the program's execution. Unconditional and conditional instructions prompt a decision and, perhaps, a branch to another part of the program or to a subroutine. In Figure 2.12, statements 10-40, 80, 110, 130, 140, 210, 270, and 340 are control instructions.

Unconditional Instructions. Statements 10 through 40 are unconditional branch instructions. An unconditional branch instruction disrupts the normal sequence of execution by causing an unconditional branch to another part of the program or to a subroutine. In statements 10-40, the branch is from the driver module to a subroutine. The CALL statement works in conjunction with the RETURN statement to branch to another location, then RETURN control back to the statement following the CALL. For example, the CALL at statement 10 passes control to the "Input Data" module (1.1) at statement 50, then the RETURN at statement 120 passes control back to the driver module at statement 20.

Another unconditional branch instruction, very popular before structured programming, is the GOTO (pronounced, go too) instruction. The GOTO statement causes control of execution to "go to" another portion of the program. A GOTO instruction is placed at statement 170 so that once

the hourly pay is calculated, control is passed directly to the RETURN statement in order to bypass that portion of module 1.2 that deals with commission employees.

The GOTO at statement 170 was included for demonstration purposes, but in an actual program it should have been avoided when possible. Programming gurus advocate that the GOTO statement be used sparingly. Excessive use of GOTOs destroys the logical flow of the program and makes it difficult to divide the program into modules. Some companies simply do not permit the use of GOTO statements at all, thus the term GOTO-less programming. The theory behind GOTO-less programming is that programmers tend to use GOTOs to get out of a "programming corner" instead of applying good sound logic. If GOTOs are not allowed, then programmers are encouraged to divide the program into modules and to make judicious use of the three control structures in Figures 2.5-2.7. For example, to avoid the GOTO statement (160) in the example, the "compute gross earnings" module (1.2) could have been subdivided into Modules 1.2.1 (hourly computation) and Module 1.2.2 (commission computation).

The END instruction at statement 130 terminates program execution. Control is passed to the END instruction from statement 110 when the end-of-file marker is detected.

Conditional Instructions. Statements 80 and 110 are conditional branch instructions and are generally referred to as IF statements: If certain conditions are met, then a branch is made to a certain part of the program. The conditional branch at statement 80 causes the program to "loop" until the employee record read is for either an hourly or a commission employee, or the end-of-file marker is reached. The sequence of instructions, statements 50 through 80, comprise a DOUNTIL loop. The IF at statement 110 causes the program to terminate if there are no more employees to be processed. Each programming language offers one or more specialized instructions for the express purpose of creating loops.

In statement 140, the salary code is checked. IF the salary code is "commission", then a branch is made to statement 180 and processing is continued.

Data Transfer and Assignment

Data can be transferred internally from one primary storage location to another. In procedure-oriented languages, data are transferred or

"moved" by assignment instructions. These instructions permit a string constant, also called a literal value, such as "The net pay is", or a numeric value, such as 234, to be assigned to a named primary storage location.

In a program, a primary storage location is represented by a variable name (e.g., PAY, HOURS, NET). A variable name in a program statement refers to the contents of a particular primary storage location. For example, a programmer may use the variable name HOURS in a computation statement to refer to the numeric value of the hours worked by a particular employee.

Data are transferred internally by simply equating the contents of two variables. In Figure 2.13, for example, an employee's salary is read into primary storage and stored in a location named SALARY. A program statement, PAY = SALARY, then transfers the contents of the primary storage location named SALARY (3000) to the primary storage location named PAY. The previous contents of PAY (2550) is destroyed and replaced with the contents of SALARY. The contents of SALARY, though, remains the same.

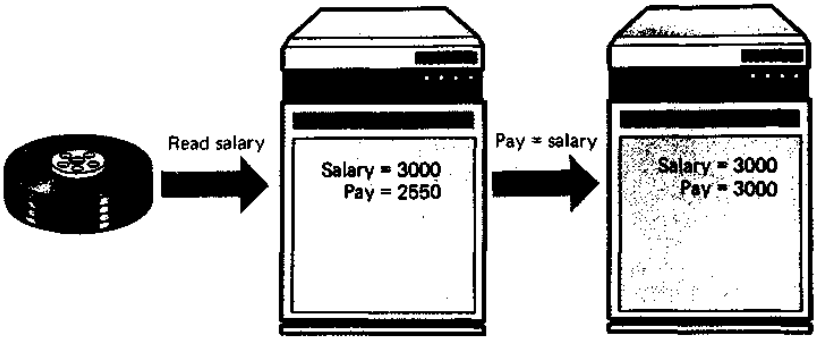


Figure 2.13 – Internal Data Transfer

In statement 260 of Figure 2.12, NET = PAY – DEDUCT, the arithmetic result of subtracting DEDUCT from PAY is stored in NET.

Format. Format instructions are used in conjunction with input and output instructions; they describe how the data are to be entered or outputted from primary storage. When the READ at statement 50 retrieves an employee's record from secondary storage, it is loaded to primary storage as a string of characters. The format instruction enables the

program to distinguish which characters are to be associated with the variables EMPLOYEE-NAME, SALARY-CODE, and so on.

On output, format instructions print headings on reports and present data in a readable format. For example, PAY may be computed to be 324750; however, on output, you would want to "edit" 324750 and insert a dollar sign, a decimal point, and perhaps a comma, for readability ($3,247.50). This is called editing the output.

With these few types of instructions, you can model almost any business or scientific procedure, whether it be sales forecasting or guiding rockets to the moon.

2.5. Writing Programs

If you are writing programs to implement an information system, then any programming assignment would be accompanied by the systems specifications and program descriptions.

Remember, writing a program is a project within itself. The following steps are followed for each programming project.

- Step 1. Describe the problem.
- Step 2. Analyze the problem.
- Step 3. Design the general logic of the program.
- Step 4. Design the detailed logic of the program.
- Step 5. Code the program.
- Step 6. Test and debug the program.
- Step 7. Document the program.

Step 1. Describe the Problem. A problem might be for example to write a program that accepts numeric quiz scores and assigns a letter grade. Another problem might be to write a program that identifies and prints the names of customers whose accounts are delinquent. Your instructor will probably define the problem for programs that you might do as class assignments.

Step 2. Analyze the Problem. In this step, you break the problem into its basic components for analysis. Remember, "Divide and conquer". Although different programs have different components, a good place to start with most programs is to analyze the output, input, processing, and file-interaction components. You would then identify important considerations in the design of the program logic. At the end of the problem

analysis stage, you should have a complete understanding of what needs to be done and a good idea of how to do it.

Step 3 and 4 Design the General and Detailed Logic of the Program. Next you have to put the pieces together in the form of a logical program design. Like the information system, the program is also designed in a hierarchical manner, or from the general to the specific.

The General Design (Step 3). The general design of the program is oriented primarily to the major processing activities and the relationships between these activities. The structure chart of Figure 2.1 and the flowchart of Figure 2.4, both discussed earlier in this chapter, illustrate the general design of a weekly payroll program to compute and print pay checks. By first completing a general program design, you make it easier to investigate alternative design approaches. Once you are confident of which approach is best, a more detailed design may be completed.

The Detailed Design (Step 4). The detailed design results in a graphic representation of the program logic that includes all processing activities and their relationships, calculations, data manipulations, logic operations, and all input/output. Figure 2.8, discussed earlier, contrasts general and detailed program design.

It is best to test the logic of a program in graphic format (e.g., flowchart or data flow diagram) before you code the instructions. If you rush into Step 5 (Code the Program), you will spend a lot of unnecessary time backtracking to fix the things you overlooked. Resist the urge to start coding before you have the logic of the program firmly fixed in your mind- and on paper. After all, if you don't have time to do it right the first time, how could you possibly have time to do it over?

Peer Review of Design. More often than not, programming is a team effort. In programming, it is true that “two heads are better than one”. For this reason, programmers often ask their colleagues to evaluate their logic before proceeding to code the program. This can be done formally, through a structured walkthrough, or informally.

Whether you are reviewing the design of your own program or somebody else’s, you would look for such attributes as ease of understanding, soundness of logic, and attainment of the required objectives.

A completed program is more a beginning than an end. Once the program goes into production, it must be constantly revised to meet the changing needs of the company. A poorly designed program can be a

nightmare to the programmers who inherit the maintenance responsibility for it over the life of the system. A peer review, such as a structured walkthrough, is a good way to eliminate bad programs and make long-term maintenance a more enjoyable task.

A team of programmers is usually assigned to work on big projects. The programming team meets as a group at least once per week to coordinate efforts, discuss problems, and report on individual and team progress.

Step 5. Code the Program Whether you "write" or "code" the program is a matter of personal preference. In this context, the terms are the same. In Step 5, the graphic and narrative design of program development Steps 1 to 4 is translated into machine-readable instructions, or programs. If the logic is sound and the design documentation (e.g., flowcharts, pseudocode, and so on) is thorough, then the coding process is relatively straightforward.

The best way to write a program is to work directly from the design documentation and compose the program interactively at a workstation or PC. Remember, programs are much easier to write when broken down into several small and more manageable modules. Programmers have shown time and time again that it takes less time to code ten 50-line modules than it does to code a single 500-line program.

When you write a program, you should have appropriate documentation on hand. This documentation may consist of some or all of the following:

- The data dictionary (with standardized names for variables, e.g., NET, HOURS)
- The coding scheme for coded data elements
- The file layouts and data base schemas
- Printer and video display layouts
- Data entry specifications
- The program design documentation (e.g., HIPO charts, flowcharts, program descriptions, and so on)

You may not need to write original code for every programming task. Many organizations maintain libraries of frequently used program modules, called reusable code. For example, several programmers might use the same reusable code for an input subroutine.

Step 6. Test and Debug the Program. Once the program has been entered into the system, it is likely that you will encounter at least one of

those cantankerous bugs. A bug is an error in program syntax and logic, Ridding a program of bugs is the process of debugging.

Syntax Errors. Debugging a program is a repetitive process, whereby each successive attempt gets you one step closer to a working program. The first step is to eliminate syntax errors or diagnostics; you get a syntax error when you violate one of the rules for writing instructions (e.g., placement of parentheses, spelling of commands, and so on). The diagnostics on the first run are mostly typos (e.g., REED instead of READ). Most compilers and interpreters identify the number of the statement causing the diagnostic and give you an error message. As you will quickly find out, if you haven't already, error messages are not always totally explanatory. The error message points you in the right direction, but some extra effort may be required to isolate the exact cause of the error.

Logic and I/O Errors. Once the diagnostics have been removed, the program can be executed. A diagnostic-free program is not necessarily a working program. You now have to debug the logic of the program and the input/output formats. To do this, you need to create test data and, perhaps, a test data base so you know what to expect as output. For example, suppose you write a program to average three grades and assign a letter grade. If your test data are 85, 95, and 75, then you would expect the average to be an 85 and the letter grade to be “B”. If the output is not 85 and “B”, then there is a bug in the program logic.

A program whose logic is sound might have input or output formats that need to be “cleaned up” to meet layout specifications. Suppose your output looked like this:

```
THE LETTER GRADE ISB
```

and the layout specs called for this:

```
THE LETTER GRADE IS B
```

Then you would need to modify the output format to include a blank space between IS and the letter grade.

Most of the interactive programming languages have software debugging aids. One of the most helpful aids, especially for finding logic errors, is the trace. When you ask for a trace, you get a sequential log of the order in which statements or sections of the program are executed. The

trace also shows you which branches were taken during execution. By comparing the actual sequence of execution against the expected sequence, you can usually isolate the error.

Test Data. Test data are an integral part of the test procedure. Test data are made up so that all possible circumstances or branches within the program are tested. Good test data contain both valid and erroneous data. It's always a good idea to deliberately introduce erroneous data to see if error routines are working properly. For example, in a program that averages grades, you might wish to include an error routine that questions grades greater than 100. To test this routine, you simply enter a grade of, say, 108. If the program works properly, the error routine will detect the erroneous data and display an error message. A good programmer lives by Murphy's Law, which assumes that if it can happen, it will! Don't assume that whoever uses your program will not make certain errors in data entry.

A program may be bug-free after unit testing, but eventually it will have to pass systems testing, where all programs are tested together. Thorough testing is essential to quality information systems, otherwise a "bug" might fly up and "byte" you at the worst possible time. It has happened before.

Step 7. Document the Program. When you write a program in a college course, you turn it in, it's graded, and then it's returned. Your effort contributes to your knowledge and expertise, but in all probability the program you wrote will not be used again. In a business environment, however, a program that you write may be used every day-for years!

Over the life of the system, procedures and information requirements change. For example, because the social security tax rate is revised each year, certain payroll programs must be modified. To keep up with these changes, programs must be periodically updated, or maintained. Program maintenance can be difficult if the program documentation is not complete and up to date.

The programs you write in college are not put into production and are, therefore, not maintained. You may ask, "Why document them?" The reason is simple. Good documentation now helps to develop good programming habits that will undoubtedly be carried on in your future programming efforts. Documentation is part of the programming process. It's not something you do after the program is written.

In business, you may or may not be responsible for maintaining your own programs. In all probability, other programmers will eventually be

given maintenance responsibility for your programs. A well-documented program will make life much easier for those who must maintain it. You might say, “Well, I know the program backward and forward and don’t really need much documentation”. You might be intimately familiar with a program that you have just completed, but what about six months and 20 programs from now? You would be surprised at how much we humans forget about our own creative work. A good program documentation package includes the following items:

§ *Program Title.* A brief descriptive title (e.g., PRINT _ PAYROLL _ CHECKS).

§ *Language.* Language in which program is written (e.g., COBOL, BASIC).

§ *Narrative Description.* A word description of the functions performed.

§ *Variables List.* A list containing the name and description of each variable used in the program.

§ *Source Listing*. A hard-copy listing of the source code.

§ *Detailed Program Design.* The flowcharts, decision tables, and so on.

§ *Input/Output Layouts.* The printer and workstation display layouts; examples of hard-copy output (e.g., payroll check).

§ *Frequency of Processing.* How often the program is run (e.g., daily, weekly, on-line).

§ *Detailed Specifications.* The arithmetic computations, sorting and editing criteria, tables, control totals, and so on.

§ *Test Data.* A test package that includes test data and expected results. The test data are used to test and debug the program after each program change.

Some of these documentation items can be included in the actual program as internal documentation. Descriptive programmer remarks throughout the program make a program easier to follow and to understand. Typically, the program title, a narrative description, and the variables list would also be included as internal documentation.

A programmer in a project team might be responsible for one or a dozen programs. However, each program evolves through this seven-step process.

Task II. Fill in the gaps using the list of words given below.

1. Programs are written in _____, or independent tasks.
2. The combination of symbols and flow lines describe the _____ of the program or system.
3. A variable name in a program statement refers to the _____ of a particular primary storage location.
4. The first step in debugging of a program is to eliminate syntax errors or _____.
5. "Subtotal amount" is a _____ constant.
6. You get a _____ when you violate one of the rules for writing instructions.
7. A program module should begin and end with the same instructions each time it is executed, thus transfer of control into and out of a particular program module occurs only at the _____ points.
8. _____ are now available that permit you to interactively create flowcharts, structure charts, and data flow diagrams on a display screen.
9. In structured programming, each program is designed with a _____ that calls _____ as they are needed.
10. _____ is not something you do after the program is written. It is a part of the programming process.

(contents, syntax error, driver module, diagnostics, logic, entry and exit, modules, documentation, subroutines, software packages, string)

Task III. Decide whether the following statements are true or false in relation to the information in the text. If you think a statement is false, change it to make it true

1. The software for an electronic spreadsheet is contained in a single program.
2. Computer programs direct the computer to perform calculations and manipulate data.
3. The effectiveness of structured programming is still a matter of debate.
4. Flowcharting is used primarily for program design and rarely for systems design.
5. IF-THEN-ELSE logic is associated with the selection structure.

6. There is a direct relationship between the number of GOTO instructions in a program and how well the program's design is structured.
7. Frequently used program modules are called reusable code.
8. Once a program has been unit tested, no further testing is required.

Task IV. Answer these questions about the text.

1. How do we direct computers to perform calculations and manipulate data?
2. What can provide solutions to particular problems?
3. To solve the problem you must use your creativity. What does it mean?
4. How are the most effective problems designed?
5. Explain the principles of structured programming.
6. What kind of techniques are commonly used to represent systems and programming logic?
7. Name and illustrate the three basic program control structures.
8. What are the benefits of structured programming?
9. What is the purpose of a test-on-condition instruction?
10. Where is the test-on-condition placed in a DOWHILE loop? In a DOUNTIL loop?
11. What does pseudocode represent?
12. Describe the decision table.
13. In a decision table, in which quadrant is the condition stub? The action entries?
14. What is a computer program made up of?
15. Name five classifications of instructions.
16. In the writing of a program, what seven steps are followed?
17. What documentation should you use when writing a program?
18. Describe the characteristics of good test data.
19. What items does a good program documentation package include?

Task V. Discuss the following matters.

1. How a “team effort” relates to egoless programming?
2. The rationale for the “divide and conquer” approach to programming.
3. What is the rationale for completing a general design of a program’s logic before completing a detailed design?
4. Justification for the extra effort required to fully document a program.