PASCAL Programming Fundamentals

P. S. GROVER

ALLIED PUBLISHERS LIMITED

Regd. Off. : 15 J.N. Heredia Marg, Ballard Estate, Mumbai 400001
Prarthna Flats (2nd Floor), Navrangpura, Ahmedabad 380009
3-2-844/6 & 7 Kachiguda Station Road, Hyderabad 500027
16-A Ashok Marg, Patiala House, Lucknow 226001
5th Main Road, Gandhinagar, Bangalore 560009
1/13-14 Asaf Ali Road, New Delhi 110002
17 Chittaranjan Avenue, Kolkata 700072
81 Hill Road, Ramnagar, Nagpur 440010
751 Anna Salai, Chennai 600002

First published, 1988
© P.S. Grover, 1988
8th Reprint, 2001

ISBN 81-7764-193-X

Published by Sunil Sachdev and printed by Ravi Sachdev at Allied Publishers Limited, Printing Division, A-104 Mayapuri, Phase-II, New Delhi - 110 064
8/2001

Contents

Chapter 1

Introduction

A computer is an information processing machine. It can perform arithmetic operations (addition, subtraction, multiplication and division) and take logical decisions. It has a memory and can store lot of information. The stored information may be retrieved, moved and operated upon as desired. Computations are done at an extremely fast speed with complete reliability and accuracy. The speed of execution of operations by modern computers ranges from several hundred million operations per second for fast computers to tens of thousands of operations per second for slow computers.

1.1 Computers

There are basically two types of computers—analog and digital.

(a) The analog computer accepts, processes and generates continuous (unbroken) data. Computations are carried out with physical quantities, such as length, voltage, current, etc. Slide rule, voltmeter, ammeter, potentiometer, are examples of analog devices. You know that when current is passed through an ammeter, the deflecton of the needle indicates the amount of current passing. Deflection is more for higher currents and less for lower currents. Now current and deflection are both continuous quantities. The ammeter receives current (input) and gives deflection (output) after detecting current (processing of input). Thus, ammeter is an analog device. Analog computers use this principle, though they are much more complicated and can perform sophisticated processing of data.

Analog computers are comparatively slow and less accurate. They are designed for special applications only. We cannot use a given analog computer for all purposes.

(b) The digital computer accepts, processes and produces discrete (discontinuous) data. Computations are done with discrete quantities, such as numerical digits. Usual facit machines, electronic calculators are simple examples of digital devices.

Digital computers are much faster than analog computers and the computations are far more accurate. They come in various sizes-starting from pocket size to large systems which occupy few normal-sized rooms. Digital computers can be designed either for special or for general purposes.

Another type of computer which combines the salient features of both analog and digital computers, is referred to as Hybrid computer. It is faster than analog but much slower than digital computer. Hybrid computers find applications in special areas only.

Normally, when we speak of a computer, it is understood as a digital computer. Nowadays these are the most widely used machines.

1.2 Computer Systems

A computer system consists of a computer and supporting devices for input and output of data. The data to be processed are supplied to the computer with the help of input devices. The processing unit performs the desired operations on the information and the results of calculations/processing are obtained on the output devices. Several types of input/output devices can be attached to the computer. A computer consists of electronic circuits only, while the input/output devices have both electronic and mechanical components. A representative organization of a computer system may be as shown in Fig. 1.1.

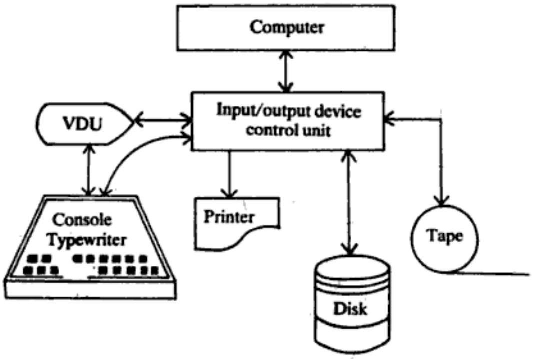


Fig. 1.1: Organization of a computer system

The input device supplies data, as obtained from us, to the computer. Most commonly used input devices are: teletypewriter and cathode ray tube (In the past, it used to be card reader but not now). We specify the data in a form which we use in our everyday lite, that is, in the numeric and alphabetic form. These are converted into the form which the computer can 'understand'. Data are stored in the computer in binary form (see later). (The conversion to the binary form is performed with the help of electronic circuits called Encoders). After the computer has processed the data, results are tained in the human readable form on the output devices, such as printer, visual display unit (VDU). Computer data (as stored inside it) is changed into this form by electronic circuits called Decoders.

In addition to encoders and decoders, there are other electronic circuits also which perform several functions, such as correct transfer of data from the input device to the computer and from the computer to the output device, regulate flow of data, act as temporary data storage (buffering), etc. This is done with additional electronic circuits, referred to as control input/output unit. All input/output devices are connected to the computer via control units.

The computer does all the computing and data processing work. Its components are:

(a) Arithmetic and logical unit (ALU)
(b) Control unit (CU)
(c) Main memory unit (MMU).

Information is transferred to and from among these units for all processing and computing work as indicated by arrows in Fig. 1.2.

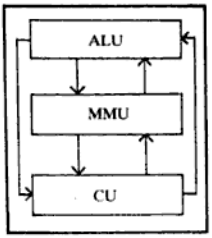


Fig. 1.2: Information flow in the computer

Artithmetic and Logical Unit

This unit consists of a complicated electronic circuits designed using the concepts of Boolean algebra. All arithmetic operations-addition, subtraction, multiplication, division and logical operations-comparison, decision, etc. are performed by this unit.

Control Unit

The control unit also consists of electronic circuits. It acts as a supervisor in a computer system. It obtains instructions from the main memory, interprets them, decides the action to be taken and directs other units to execute them. It keeps check on correct information flow in the computer system. Normally, the instructions are executed sequentially (one after another) in the machine. The control unit also provides the facility to alter this sequence.

The ALU and the CU are also referred to as Central Processing Unit (CPU).

Main Memory Unit

This unit stores all the data which are to be processed and the program instructions for carrying out the processing/computing work. The main memory is also referred to as primary or main storage. It is extremely fast (high speed memory). Information can be entered/retrieved at random from this memory.

Commonly used memories have been magnetic cores (older systems) and semiconducting elements (modern systems). With these, it is possible to have very large and fast memories. Semiconductor memory consists of electronic circuits prepared on silicon chips. The electronic circuit is called a Flip-flop. A flip-flop circuit can generate either 1 or 0, that is, it is a two-state element. A flip-flop is also called a Storage Cell. Thousands of these storage cells can be prepared on a single silicon chip. Due to this, the physical size of the semiconductor memories is very small. Moreover, their cost is decreasing every year as the fabrication technology is advancing. The common type of semiconductor storage devices are: Random Access Memory (RAM) and Read only Memory (ROM). Besides these, there are other forms of semiconducting memories as well.

In RAM, information may be read or written into the memory at random. It is also called a Read/Write memory. It is a volatile memory, that is, information stored there is lost when the electrical power to the circuit is switched off. Normally, user programs and data are stored in RAM.

In ROM, information is written permanently into the memory. It cannot be changed easily. Data can be read from the memory but cannot be written there. This is why the name ROM. Moreover, it is a non-volatile memory. ROMs are normally used to store information that the computer may need frequently for its own operation.

1.3 Computer Generations

Several types of computers, having wide range of characteristics have been designed. The design, speed, size and performance of computers have been changing continuously. Due to this, it has become customary to divide computers into what has come to be known as "generations". Broadly speaking, following are the generations of the computers.

First Generation (~ 1946-1959)

The computers of first generation used vacuum tubes. They were bulky and slow. Their memory was limited and used punched-card and punched paper tape for input and output of data. These machines used low-level programming languages and involved manual controls. They were special purpose machines with limited applications. Examples of first generation computers include ENIAC, EDSAC, UNIVAC I, IBM 650.

Second Generation (~ 1959-1965)

The second generation computers are characterized by their use of tiny transistors as active elements. Due to this, these were compact and substantially smaller in size. They required less power to operate. They were much more reliable as compared to the first generation computers. Second generation computers had more speed (about 10^6 operations per second), larger memory and faster input/output devices. They accepted procedure-oriented languages, such as Fortran, Cobol and other utility programs.

Several companies started manufacturing computers. Systems were designed for special applications, such as business and scientific data processing. Prominent second generation computers have been IBM 1401, IBM 7090/7094 series, IBM 1620, Burroughs B5000, CDC 3600, GE 635, Honeywall 400 series, UNIVAC III, and several others.

Third Generation (~ 1965-1975)

The third generation computers used integrated circuits (electronic circuits designed on silicon chips) instead of transistors. The size of such circuits is hundreds of times smaller than the transistor circuit size. Moreover, the associated electronic circuitry is also reduced in dimensions many times. This leads to several advantages: (i) small size and increased processing speed ($\approx 10^8$ operations/second), (ii) more reliability and higher accuracy, (iii) easy maintenance and simple repair requirements. Moreover, these machines have very large storage capacity. Faster and more versatile input/output devices, such as video display, graphic terminals, plotters, magnetic disks, drums, tapes, etc. may be used with them. They have highly sophisticated operating systems, application software and packages.

Third generation computers are mostly general purpose, that is, they may be used for processing business, scientific or text-oriented problems. Some examples of third generation systems are IBM 360 series, Burroughs 6700/7700 series, CDC 6000/7000 series, digital equipment PDP-8/11 series, UNIVAC 1108/9 series, ICL 1900/2900 series, and so on.

Fourth Generation (~ 1972-1982)

The fourth generation computers use very large scale integrated (VLSI) circuits in their design. As compared to the third generation systems, these systems possess much larger computing powers. They have extremely large memories and are very versatile. In addition to conventional input/output devices, other minicomputers, magnetic ink readers, laser printers, optical readers, audio response terminals, etc. can be attached to them. Some examples of fourth generation computers are IBM 370, AMADAHL 470, CRAY XMP, CYBER and many other systems.

Fifth Generation and Beyond (~ 1981 onwards)

The computer technology has made phenomenal progress, starting from first to fourth generation systems. Computers have been used to solve almost any type of problem whose algorithm can be described explicitly. However, still there are areas, where computers have not been used successfully. This is because the problems in those areas involve human intelligence, that is, reasoning, understanding, drawing inferences, adapting to new situations, making out relationship between facts, discovering meanings, recognizing truth, and so on. Attempt is being made to incorporate such characterisitics into computers and computer programs. Such computers (though not commercially available as yet) have been said to belong to Fifth generation. Lot of work is being done on the hardware and software of such systems. These are also known as Knowledge Information Processing Systems (KIPS). They will have knowledge bases and are expected to be able to draw inferences from the knowledge and solve problems as humans do. These are the goals and their successful completion may completely revolutionize the computer field.

1.4 Bit, Byte, Word and Number Systems

Binary number system is used for information representation and storage in computers. This system uses two digits: 0 and 1. Any object which can assume two states can be used to represent these digits. As for example an electric switch, an electric bulb, a magnetic core or some electric circuit, such as flip-flop, and so on.

A magnetic core can be magnetized either clockwise or anticlowise. We can say that the clockwise state of magnetization represents 1 and the anticlockwise state of magnetization indicates 0.

Thus, a collection of magnetic cores can be used to store binary digits. A binary digit is also called a Binary bit. Thus, each core represents a binary bit.

Binary bits are grouped together. A collection of 4 bits is called a Nibble, while a group of 8 bits is referred to as a Byte. Normally, a byte is treated as a single memory location. It generally represents a single character of information. As for example, if we wish to store the character A in the computer memory, a byte (or 8 bits) may be required. Further bytes are grouped together to form a bigger unit. This unit is referred to as a Word. A word may consist of a single nibble (4-bit word), two nibbles (8-bit word), 3 nibbles (12-bit word), 4 nibbles/2 bytes (16 bit word) and so on. This depends on the make of the computer. The size of the computer memory is commonly expressed in terms of bytes/words. As for example, it is common to say that the memory size is 64 KB (kilo byte), 128 KB, 256 KB, and so on. Here $k = 2^{10} = 1024$ and indicates kilo.

A memory of 16 KB will have $16 \times 1024 \times 8 = 131072$ bits. Thus a semiconductor memory should have these many flip-flop circuits as each flip-flop represents one binary bit.

Nowadays, very large memories are also available. Memory sizes in the range 1—8MB (or higher) are common. (MB = million bytes). Moreover, computers are available with different word sizes such as 8—, 16—, 32—, 64—, 96-bits and so on. Larger word size affords several advantages :

- bigger main memory size

- larger instruction set and more versatile commands
- higher speed
- better data processing capabilities
- more sophisticated software systems
- use of a variety of input/output and peripheral devices
- better control and utilization of the different hardware computer system resources, such as memories, data transfer channels, disks/drums, etc.
- implementation of several efficiency and better system utilization techniques, such as multiprogramming, time-sharing, multiprocessing, and so on,
- higher accuracy of floating point computations

Many of the concepts, introduced above, will get clarified as we go along.

Number Systems

You are all familiar with decimal number system. It consists of digits 0, 1, 2, ..., 9 which form its basis and all numbers are formed by a combination of these. Other number systems commonly used are binary, octal and hexadecimal. Numbers expressed in one system are easily changed into the other. Table 1.1 gives the equivalence among the four systems. These are positional number systems and the value at each position is shown in Table 1.2, starting from right to left. We shall describe briefly the number systems and explain conversion from one system to another.

(A) *Binary Number System*

The two digits of binary number system are 0 and 1. All numbers in this system are designed using these digits only. Like decimal, binary system is also positional, that is, the digit takes a different value according to the position it occupies. Let us consider the binary number : 1. It stands for $1 = 2^0 = 1_{10}$. 10 is the subscript. It indicates decimal. Thus 1_{10} implies 1 in decimal system. Now consider

11

It stands for

$$1 \times 2^1 + 1 \times 2^0 = 3_{10}$$

that is, it is equivalent to 3 in decimal system.
Now look at

1 1 0 1

It implies

$$\begin{aligned}1101 &= 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0\\ &= 8 + 4 + 0 + 1\\ &= (13)_{10}\end{aligned}$$

Thus, a string of 1's and 0's is used to represent a number of binary system. The binary equivalent of the first 17 decimal numbers are indicated in Table 1.1.

Table 1.1: Equivalence between decimal and other systems

Decimal Number	Binary Number	Octal Number	Hexadecimal Number
0	00	0	0
1	01	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11

Table 1.2: Number systems and positional values

Position →	Five	Four	Three	Two	One
Binary	2^4	2^3	2^2	2^1	2^0
	16	8	4	2	1
Octal	8^4	8^3	8^2	8^1	8^0
	4096	512	64	8	1
Hexa	16^4	16^3	16^2	16^1	16^0
	65536	4096	256	16	1

Conversion of binary numbers to decimal form was studied above. Let us see the reverse process. The method given below may be used for small as well as for large numbers.

Take the decimal integer number. Divide it by 2, say the quotient is Q_1 and remainder R_1. Then divide Q_1 by 2. Now, let the quotient be Q_2 and remainder R_2. Further, divide Q_2 by 2. Suppose, a new quotient is Q_3, and remainder is R_3. Continue dividing the new quotient by 2 and getting new remainders till the quotient is 0. Say, the remainders for successive divisions are R_1, R_2, $R_3, \ldots, R_{n-1}$, R_n. Then the binary representation of the given decimal number will be

$$R_n R_{n-1} \ldots\ R_4 R_3 R_2 R_1 \tag{i}$$

That is, the binary number is obtained by writing the remainders in the reverse order.

We explain this procedure by an example. Say, we want to find the binary equivalent of the decimal number 85. Proceed as follows:

Quotients	Remainders
$85/2 = 42\ (Q_1)$	$1\ (R_1)$
$Q_1/2 = 42/2 = 21\ (Q_2)$	$0\ (R_2)$
$Q_2/2 = 21/2 = 10\ (Q_3)$	$1\ (R_3)$
$Q_3/2 = 10/2 = 5\ (Q_4)$	$0\ (R_4)$
$Q_4/2 = 5/2 = 2\ (Q_5)$	$1\ (R_5)$
$Q_5/2 = 2/2 = 1\ (Q_6)$	$0\ (R_6)$
$Q_6/2 = 1/2 = 0\ (Q_7)$	$1\ (R_7)$

Thus, the binary number, according to prescription (i) will be

$$\begin{array}{ccccccc} R_7 & R_6 & R_5 & R_4 & R_3 & R_2 & R_1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array} \quad \text{(Binary number)}$$

You can verify that this binary number is equivalent to 85 as

1 0 1 0 1 0 1

$$\begin{aligned}
1 \times 2^0 &= 1 \times 1 = 1 \\
0 \times 2^1 &= 1 \times 1 = 0 \\
1 \times 2^2 &= 1 \times 4 = 4 \\
0 \times 2^3 &= 0 \times 8 = 0 \\
1 \times 2^4 &= 1 \times 16 = 16 \\
0 \times 2^5 &= 0 \times 32 = 0 \\
1 \times 2^6 &= 1 \times 64 = 64 \\
&\overline{\qquad 85}
\end{aligned}$$

Thus

$$(85)_{10} = (1010101)_2$$

Now we look at the following example which gives the binary equivalent of 24.

Quotients	Remainders
$24/2 = 12\ (Q_1)$	$0\ (R_1)$ •
$Q_1/2 = 12/2 = 6\ (Q_2)$	$0\ (R_2)$

$Q_2/2 = 6/2 = 3\,(Q_4)$	$0\,(R_3)$
$Q_3/2 = 3/2 = 1\,(Q_4)$	$1\,(R_4)$
$Q_4/2 = 1/2 = 2\,(Q_5)$	$1\,(R_5)$

Thus, the binary representation of 24 is 11 000 or

$$(24)_{10} = (11000)_2$$

You can verify that this number is equivalent to 24 by the procedure explained earlier.

In practice, we can have integers or mixed numbers. Examples of mixed numbers are 12.65, 16.018, 211.101, and so on. (Remember, mixed numbers have integral part and a fractional part). In the number 12.65, the integral part is 12 while the fractional part is 0.65. We can find the binary representation of the mixed numbers also. Now it will be necessary to apply the conversion to the integral and fractional parts separately.

The conversion of integral part is to be performed as explained above. The number $(12)_{10} = (1100)_2$. What is $(0.65)_{10} = (?)_2$. The conversion rule is:

Take the fractional part and multiply it by 2 (base). Set aside the integral part (it will be either 1 or 0) and again multiply the fractional part by two. Do this repeatedly till the fractional part is zero. The sequence of integral parts (that is, 0s and 1s), thus generated, along with the decimal point in the left most position, gives the required binary representation of the decimal fractional part.

Using this rule, the equivalent form of 0.65 can be obtained as

Integer Part	Fractional Part	
I_1 1	.30	1.30 ← .65×2
I_2 0	.60	0.60 ← .30×2
I_3 1	.20	1.20 ← .60×2
I_4 0	.40	0.40 ← .20×2
I_5 0	.80	0.80 ← .40×2
I_6 1	.60	1.60 ← .80×2
I_7 1	.20	1.20 ← .60×2

The representation of .65 is

$$\begin{aligned}(.65)_{10} &= .I_1\, I_2\, I_3\, I_4\, I_5\, I_6\, I_7 \\ &= (.1010011\ldots\ldots)_2\end{aligned}$$

We observe that the process of multiplying by 2 and getting the integer part, till the fractional part is zero, can become inordinately long. This is a drawback of the binary system as regards the representation of decimal fractions is concerned. The binary representation is truncated after a certain number of places, depending on the word size of the computer. It is easy to convert .1010011 back to decimal form as

$$\begin{aligned}.1010011 &= 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} + 0\times2^{-4} + 0\times2^{-5} + 1\times2^{-6} + 1\times2^{-7} \\ &= 0.5 \quad + 0 \quad + 0.125 \quad + 0 \quad + 0 \quad + .015625 + .0078125 \\ &= (.6484375)_{10}\end{aligned}$$

Hence $(12.65)_{10} = (1100.1010011)_2$

In order to get better accuracy and approach .65, more terms, such I_8, I_9, ... will have to be included. Thus, decimal conversions are the source of truncation errors in floating point calculations in computers.

(B) *Octal and Hexadecimal Systems*

(a) *Octal System*

Octal number system has the base 8 and consists of the digits 0,1,2,3,4,5,6,7. Like binary and decimal, octal system is also a positional number system. Table 1.1 gives the equivalence of decimal, binary and octal numbers upto 17, while Table 1.2 shows the positional value. The reader should note that upto 7, both octal and decimal numbers are same. Represenation differs beyond this. Transformation from one system to the other is similar to as was for binary-decimal-binary. As an illustration, let us consider the octal number $(27)_8$. We want its value in decimal system:

$$\begin{aligned}(27)_8 &= (?)_{10}\\ (27)_8 &= 2\times8^1 + 7\times8^0\\ &= 16+7\\ &= (23)_{10}\end{aligned}$$

Further

$$\begin{aligned}(305)_8 &= 3\times8^2 + 0\times8^1 + 5\times8^0\\ &= 192 \quad + 0 \quad\quad + 5 \quad\quad = (197)_{10}\end{aligned}$$

Similarly, conversion from decimal to octal system can be performed following the procedure as indicated for decimal to binary system. Now, as the base of octal system is 8, so while converting decimal integer to octal, we need to divide the integer by 8 repeatedly until the quotient is 0. Similarly, to convert a decimal fraction to octal form, decimal fraction is multiplied repeatedly by 8 and numbers left of the decimal point are collected and arranged from left to right in the order they occurred.

Conversions binary-octal-binary are also straightward. The rule is:

Arrange the binary number digits in groups of three for octal number equivalence, starting from the right, going to left. Replace each group by its octal equivalent digit.

This rule is illustrated by the following examples.

(i) *Binary-Octal Conversion*

$$(11011)_2 = (?)_8$$

Arrange the binary digits in groups of three starting from right and going to left.

Add leading zeroes (zeroes at the left) if required, to complete group of three.

```
        011   011
         3     3    = (33)8
```

$(11100111)_2 = (?)_8$

```
(11100111)2 =  011  100  111          ←Binary-coded octal number
                3    4    7  = (347)8
```

(ii) *Octal-Binary Conversion*

Now replace each octal digit by the corresponding group of three digits.

$(24)_8 = (?)_2$

```
(24)8 =      2      4
            010    100  = (010100)2
```

$(73\,15)_8 = (?)_2$

```
(7315)8 =     7    3     1      5
             111  011   001    101   = (111011001101)2
```

Conversions binary-hexa-binary are similar to binary-octal-binary, except that now bits are to be arranged in groups of 4. The process is explained below.

(b) *Hexadecimal System*

In hexadecimal (or hexa) system, base is 16 and consists of the digits: 0,1,2,3,4,5,6,7,8,9,A(10), B(11), C(12), D(13), E(14), F(15). Hexadecimal numbers are strings of these digits. The equivalence between decimal, binary, octal and hexa system digits are given in Table 1.1 (upto 17). Again hexa system is a positional system. The value at each position in this system is shown in Table 1.2. The following examples illustrate the hexa-decimal conversion further.

$$(12)_{16} = 1 \times 16^1 + 2 \times 16^0 = 16 + 2 = (18)_{10}$$

$$(123)_{16} = 1 \times 16^2 + 2 \times 16^1 + 3 \times 16^0 = 256 + 32 + 3 = (291)_{10}$$

Conversion rules for decimal-hexa-decimal system are identical to decimal-binary-decimal or decimal-octal-decimal, as discussed earlier. Remember, now the base is 16.

(i) *Binary-Hexa Conversion*

Arrange the bits in groups of four, starting from right and going to left. Add leading zeroes (that is, 0s at the left), if needed, to complete a group of four bits. Replace each group by the equivalent hexa digit.

$(11011)_2 = (?)_{16}$

```
(11011)2 = 0001 1011
              1    B = (1B)16
```

$(11110111)_2 = (?)_{16}$

```
(11110111)2 =  1111  0111      ←Binary coded hexa number
                 F     7       = (F7)16
```

(ii) *Hexa-Binary Conversion*

Replace each hexa digit by the corresponding four-digit binary number.

$(AB)_{16} = (?)_2$

$$\begin{array}{llll} (AB)_{16} = & A & B & \\ & 1010 & 1011 & = (1010\ 1011)_2 \end{array}$$

$(3F8D)_{16} = (?)_2$

$$\begin{array}{lllll} (3F8D)_{16} = & 3 & F & 8 & D \\ & 0011 & 1111 & 1000 & 1101 \\ & \multicolumn{4}{l}{= (0011\ 1111\ 1000\ 11011)_2} \end{array}$$

The above examples illustrate how to convert the integer numbers. The procedure for converting a *binary fraction to octal (or hexa)* is similar. Now, the grouping begins at the binary point, starting from left and going to right. A binary fraction is divided into groups of three bits (or four for hexa) for octal, starting from the left of the binary point. The need may be there to add zero at the right-most side to make each group of three for octal (or four for hexa) digits. Replace each group of bits by its equivalent octal or hexa digit, as the case may be.

Further, the reader should appreciate that the binary number and its equivalent in binary-coded octal and hexa numbers have exactly the same bit arrangement. So the string of 0's and 1's stored in the computer memory unit may represent a binary number or an octal number or a hexa number in binary coded form if we group the bits in units of three (octal) or four (hexa) bits. The number of *digits* is reduced by one-third in octal representation and by one-fourth in the hexa representation. (How?)

We saw that direct conversions decimal-octal-decimal, decimal-hexa-decimal are possible. However, in applications, it is found to be more convenient and fast to use binary as intermediate base instead of converting directly. This is because simple grouping of bits yields the desired conversion.

Octal and hexadecimal systems also find applications in data communications and I/O unit interfacing. They are also used in small computers. For example, in computers with 16-bit word lengths, the hexa digits are more convenient to use. We can represent a 16-bit number with exactly four hexa digits. The hexa numbers are more compact, but are less easy to convert because of the mixture of numerals and letters.

Arithmetic calculations are generally done in binary system because they are simpler to implement electronically. When octal/hexa systems are used, they are converted to binary, computations made and then numbers converted back to octal/hexa system.

1.5 Binary Computer Codes

We have been talking about binary digits and data representation. Decimal numbers are represented by a string of 0's and 1's. How about other characters, such as A, B, P, Q, a, r, e +, ?, etc. To represent letters, special characters and numerals (collectively referred to as alphanumeric characters), we need appropriate codes. Many different codes are available. They all use binary bits because the computers can hold only binary information. Here we shall present commonly available coding schemes.

(1) *BCD Code*

BCD indicates Binary Coded Decimal. Four bits are used to represent decimal digits 0 to 9. For example, consider the decimal number 25. Its binary representation is

$$(25)_{10} = (11001)_2$$

In BCD code, binary representation is used for each of the digits in a number. Thus, the representation of 25 in BCD code is

$$(25)_{10} = \begin{array}{cc} 2 & 5 \\ 0010 & 0101 \end{array} \quad \text{(BCD code)}$$

25 will be represented by 8 bits or 00100101 in BCD code. In pure binary form, 5 bits were needed. Similarly,

$$(607)_{10} = \begin{array}{ccc} 6 & 0 & 7 \\ 0110 & 0000 & 0111 \end{array} \quad \text{(BCD code)}$$

Thus 12 bits: 011000000111 represent the decimal number 607. (What is 607 in pure binary form? Calculate the number of bits required). The reader should appreciate the difference between conversion of decimal numbers to binary form and binary coding of decimal numbers very carefully.

Four bits can be arranged in $2^4 = 16$ different possible ways. These are indicated in Table 1.3.

The first ten of these arrangements are used to represent decimal digits 0-9. The other six combinations 1010, 1011, 1100, 1101, 1110, 1111 corresponding to decimal numbers 10, 11, 12, 13, 14, 15 are not used. As the BCD representation of, say 13, will be

$$(13)_{10} = \begin{array}{cc} 1 & 3 \\ 0001 & 0011 \end{array} = 00010011 \quad \text{(BCD code)}$$

Remember, 4 bits for each decimal digit. The BCD code is also referred to as 8-4-2-1 code.

(2) *6-bit BCD Code*

This code uses 6 bits to reprsent a character. That is why it is called a 6-bit BCD code. Now $2^6 = 64$ characters can be represented. This is a sufficient number to code the decimal digits (10), capital alphabetic letters (26) and other special

Table 1.3: Place value in BCD code

Place value →	8	4	2	1	Decimal digit
	0	0	0	0	0
	0	0	0	1	1
	0	0	1	0	2
	0	0	1	1	3
	0	1	0	0	4
	0	1	0	1	5
	0	1	1	0	6
	0	1	1	1	7
	1	0	0	0	8
	1	0	0	1	9
	1	0	1	0	10
	1	0	1	1	11
	1	1	0	0	12
	1	1	0	1	13
	1	1	1	0	14
	1	1	1	1	15

sybmols (28). In this code, the four BCD numeric place positions (1,2,4,8) are retained but two extra zone positions are added. The format of a 6-bit code is

Zone bits		Numeric bits			
B	A	8	4	2	1

Table 1.4 gives the 6-bit BCD code for decimal and alphabetic characters, along with the 8-bit EBCDIC and ASCII codes.

Table 1.4: Representation of selected symbols

Character	6-bit BCD	EBCDIC	ASCII 8-bit
0	00 0000	1111 0000	1011 0000
1	00 0001	1111 0001	1011 0001
2	00 0010	1111 0010	1011 0010
3	00 0011	1111 0011	1011 0011
4	00 0100	1111 0100	1011 0100
5	00 0101	1111 0101	1011 0101
6	00 0110	1111 0110	1011 0110
7	00 0111	1111 0111	1011 0111
8	00 1000	1111 1000	1011 1000

Character	6-bit BCD	EBCDIC	ASCII 8-bit
9	00 1001	1111 1001	1011 1001
A	01 0001	1111 0001	1100 0001
B	01 0010	1100 0010	1100 0010
C	01 0011	1100 0011	1100 0011
D	01 0100	1100 0100	1100 0100
E	01 0101	1100 0101	1100 0101
F	01 0110	1100 0110	1100 0110
G	01 0111	1100 0111	1100 0111
H	01 1000	1100 1000	1100 1000
I	01 1001	1100 1001	1100 1001
J	10 0001	1101 0001	1100 1010
K	10 0010	1101 0010	1100 1011
L	10 0011	1101 0011	1100 1100
M	10 0100	1101 0100	1100 1101
N	10 0101	1101 0101	1100 1110
O	10 0110	1101 0110	1100 1111
P	10 0111	1101 0111	1101 0000
Q	10 1000	1101 1000	1101 0001
R	10 1001	1101 1001	1101 0010
S	11 0010	1110 0010	1101 0011
T	11 0011	1110 0011	1101 0100
U	11 0100	1110 0100	1101 0101
V	11 0101	1110 0101	1101 0110
W	11 0110	1110 0110	1101 0111
X	11 0111	1110 0111	1101 1000
Y	11 1000	1110 1000	1101 1001
Z	11 1001	1110 1001	1101 1010

The word INDIA will be represented in 6-bit BCD code as

```
    I       N       D       I       A
 011001  100101  010011  011001  001001
```

and the number of bits required to represent INDIA will be $6 \times 5 = 30$.
The number 607 will be represented as

$(607)_{10} = $ 6 0 7

```
        000110  000000  000111
```

that is, by 000110000000000111 in 6-bit BCD code. There are 18 bits.
Similarly, we can represent A4D9 as

A4D9 = A 4 D 9

```
        010001  000100  010100  001001
```

or 010001 000100 010100 001001 will stand for A4D9. In this system, a byte is if 6 bits.

(3) *8-bit Binary Codes*

You have seen that only 64 characters can be represented in 6-bit BCD code. This is not sufficient to represent the lower case and graphic symbols which are required in many computer applications. Due to this, 6-bit code has been extended to 8 bits. With 8 bits, it is possible to have $2^8 = 256$ different combinations. Now there are four zone bits. The format of a 8-bit code is:

Zone bits				Numeric bits			
Z	Z	Z	Z	8	4	2	1

There are two commonly used 8-bit codes. One is called Extended Binary Coded Decimal Interchange Code (EBCDIC). It was developed by IBM and is used in most IBM machines and many other computers. The other 8-bit code is referred to as the American Standard Code for Information Interchange (ASCII). This code is used in microcomputers and data communications. Table 1.4 gives the equivalent EBCDIC and ASCII codes for the digits 0—9 and characters A-Z. (See also Appendix VI). The main difference between the two codes is in the bit patterns of the zone positions. The word LOVE will be represented in these codes as:

```
   L          O          V          E
11010011  11010110  11100101  11000101  (EBCDIC)
(a string of 32 bits)

11001100  11001111  11010110  11000101   (ASCII)
(a string of 32 bits)
```

The size of byte in EBCDIC and ASCII codes is 8 bits. Now 8-bit byte has become standard and a byte is always taken as of 8 bit. Remember, a *byte represents a character.*

Binary codes find extensive use in various fields. They can be designed for any type of discrete entities. Examples are: musical notes, colours and so on. They are very useful in data communications, and several other applications as well.

We find from the above discussion that data, whether alphabetic or numeric, are represented by strings of 0s and ls. Then how does the computer differentiate between numeric and character data? This is done by instructions or commands. There are separate instructions for character data and numeric data. If an instruction, designed for character data, references a string of 0s and 1s, then this will be taken as data of character type. Similarly, if an instruction, designed for numeric data, references string of 0s and 1s, then this string will be interepretted to represent numeric data. The computer is a very precise and exact instrument and does not allow any kind of ambiguity.

1.6 Modes of Computer System Operation

The common computer system modes of operation are:

- batch
- time-sharing
- multiprogramming
- multiprocessing.

(a) *Batch Processing*

In batch mode, each user prepares his program on an off-line device (generally) and submits it to the computer centre. Several programs are read together by the computer, and they are processed sequentially, that is, one after the other. Generally, the program, the associated data, and other information are entered through punched cards. Once a program is read into the computer, the user cannot alter any statement of the program. If there is any mistake in a program, that program does not run and the next program is processed automatically. As the execution of programs takes place at a very fast speed, the batch mode is most suited for programs that are long or require large computing time. For the same reason, for small and short programs this mode is not desirable.

(b) *Time-sharing Mode*

In the time-sharing mode, the user communicates with the computer via a terminal. The commonly used terminals are: teletypewriter and visual display. Generally, the terminals are far away from the computer. They are connected to the computer by either telephone wires or microwave/satellite links. The terminal serve both as input and output devices simultaneously as programs continue to be processed while the input/output devices are busy dealing with other information. Since there may be 30/40 (or more) terminals to a computer in the time-sharing mode, several users can use the computer simultaneously, independent of each other. Therefore, it is suitable for short programs. The computer system operating in the time-sharing mode may be organized as shown in Fig. 1.3. In this figure, TERM1, TERM2, TERM3, ... are terminals. The arrows, pointing on both the sides (↔), indicate the two-way flow of information. Disk1 and Disk2 are magnetic disks which form the secondary or auxiliary storage area.

The time-sharing computer system can run programs in the batch mode as well. But the operation is rather complex. During computer operation, there is hardly any or no human intervention involved except for giving system commands. This is all managed and run by another complicated set of programs called the Operating System. These programs are supplied by the computer manufacturer and the user is never concerned with them. The present-day computers perform calculations at a tremendously fast rate ($\approx 10^7 - 10^8$ arithmetic operations per second are common). But, the input and the output

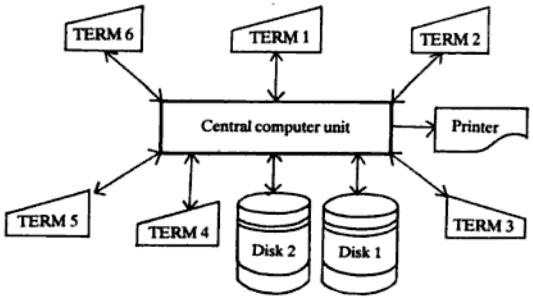


Fig. 1.3: Time-sharing mode of computer operation

being via mechanical devices, are rather slow. Therefore, the time-sharing mode is not recommended for long programs.

(c) *Multiprogramming*

The mismatch between the speed of input/output device and CPU, mentioned above, leaves some resources of the computer system under-utilized, However, if the computer system is working in the multiprogramming mode, better utilization of the available equipment can be realized.

Multiprogramming refers to keeping several programs in different parts of the main memory at the same. In this way, it is possible to keep all parts of the computer system busy. For example, suppose a given program is being executed until it demands an input/output service. Since these devices are slow compared to the CPU, the device is initiated in its function and input/output starts. Then, the CPU begins executing another different program (resident in memory) until this program asks for input/output. When this happens, the CPU may begin executing the first or third program (as the case may be), and this process continues. Thus, in multiprogramming mode, when one program is waiting for input or output operation, there is another program ready to use the CPU.

Multiprogramming is possible in both batch and time-sharing mode of computer operations.

(d) *Multiprocessing*

Multiprocessing refers to the simultaneous processing of two or more parts of the same program by two or more processing units, that is, CPUs. Now the computer system consists of more than one CPU. Generally two or more CPUs share common memory and input/output equipment. (The CPUs may have independent resources as well, but they will be closely co-operating and sharing each others resources). Multiprocessing mode of computer system operation improves the performance and reliability of the system. If one CPU fails, other CPUs can do the processing. There may be loss of efficiency, but the complete system will not be down.

It is also possible to perform simultaneous processing of different programs on a system with several CPUs (one program per CPU) sharing a common memory. This is taken as an extension of multiprogramming rather than defined as multiprocessing.

Multiprocessing computer systems are being increasingly designed now-a-days due to the easy availability of cheap VLSI circuits and microprocessors.

We have described simple concepts about computers, computer system, number systems and binary codes in this chapter. Next we go over to the discussion of problem solving on computers and introduce the reader to the basics of Pascal language.

Exercises 1

1.1. Complete the following sentences :

(a) A computer can perform .... operations and take .... decisions.
(b) The two types of computers are .... and ....
(c) The CPU of a computer system refers to .... and ....
(d) Semiconducting storage cell is actually the .... circuit.
(e) Information is written .... into ROM
(f) VLSI circuits are extensively used in .... and .... generation computers.
(g) Computers are available with common words lengths as ...., ..., ..., ... and .... bits.
(h) All number systems are ....
(i) The base digits of hexa decimal system are ....
(j) During conversion from binary to hexa representation, each group of .... bits is replaced by the corresponding .... digit.
(k) While converting decimal integer to octal, we need to .... the integer by .... repeatedly until the quotient is ....
(l) Binary codes can be designed for any type of .... entities.
(m) Multiprocessing refers to .... processing of two or more parts of the same .... by two or more .... units.

1.2. Tick the correct words in the following sentences:

(a) All computations are done by the ALU/CU.
(b) VLSI circuits are prepared on the silicon/germanium chips.
(c) Knowledge Information Processing Systems are said to belong to fourth/fifth generation computer systems.
(d) BCD/EBCDIC is an 8-bit system.
(e) A flip-flop circuit represents 1/4 binary bit(s).
(f) Conversion rules for decimal-hex-decimal system are similar/disimilar to decimal-binary-decimal system.
(g) A binary-coded number and its equivalent binary-coded octal and hexa numbers have identical/different bit arrangements.
(h) Programs are executed randomly/sequentially in batch processing mode of computer system operation.
(i) Time-sharing mode of computer operation is more suitable for short/long programs.
(j) Better utilization of computer system resources is possible with uni/multi-programming mode of computer system.

(k) Multiprogramming may/maynot be implemented with the time-sharing mode of computer operation.
(l) Operating System is a computer dependent/independent software system.
(m) The computer differentiates between numeric and character type information by appropriate instructions/data.

1.3. What is a computer? Explain briefly, the working principle of various types of computers.

1.4. Describe the basic components of a computer system. Prepare its complete diagram and explain the function of each apart.

1.5. List the various computer generations. What are the basis of such characterizations?

1.6. What are the word sizes of commonly available computers? Bring out the advantages of large word sizes.

1.7. Bring out the differences and similarities among binary, octal and hexadecimal number systems.

1.8. Explain the conversion rules for numbers for decimal-octal-decimal and binary-hexa-binary systems. Illustrate by examples.

1.9. Perform the following conversions:

(a) $(1101)_2 = (?)_{10}$
(b) $(2016)_8 = (?)_{10}$
(c) $(ABIF)_{16} = (?)_{10}$
(d) $(lE2C)_{16} = (?)_2$
(e) $(2460)_8 = (?)_{16}$
(f) $(4096)_{10} = (?)_{16}$
(g) $(1111)_8 = (?)_2$

1.10. Carry out the following transformations:

(i) $(13.50)_{10} = (?)_2 = (?)_8 = (?)_{16}$
(ii) $(127.165)_{10} = (?)_2 = (?)_8 = (?)_{16}$
(iii) $(111.011)_2 = (?)_{10} = (?)_8 = (?)_{16}$
(iv) $(123.456)_8 = (?)_2 = (?)_{16} = (?)_{10}$
(v) $(7\,AF.12C)_{16} = (?)_{10} = (?)_2 = (?)_8$.

1.11. What are the various binary computer codes? Bring out the need of having these coding schemes.

1.12. Explain the difference between binary conversion and binary coding of decimal numbers. Write the following numbers in binary and BCD codes.

(a) 13
(b) 1313
(c) 6303
(d) 1101

which is easier to write? Which one needs more bits?

1.13. Write the following characters in 6-bit, EBCDIC and ASCII codes:

(a) 6543
(b) MAN
(c) PAR77
(d) YOUR NAME
(e) A2B8

1.14. Explain the following

EBCDIC
ASCII
RAM
ROM
KIPS
Byte
Flipflop
Alphanumeric characters
Volatile and nonvolatile memories.

1.15. Describe the common modes of computer operation. How is time-sharing mode different from batch mode of operation?

1.16. Explain the differences between multiprogramming and multiprocessing. How do they help to utilize the computer system more efficiently?

Chapter 2

Problem Solving and Pascal

The term 'problem' is used to mean a task or job. Problem solving implies carrying through the task to arrive at the right solution. Computers are the best aid to solve problems. Problems arise in every field of human activity. People in different professions have different problems to solve. For example, a doctor is concerned with quick and correct diagnosis of the disease, an engineer is involved with the design of buildings, dams, machines, etc., a space-scientist may be involved with space problems and sending men to outerspace, while a buisnessman wants to run his business to maximize his profits, and so on. In fact, the list is unending. The computer can help to solve problems almost in every field. It may be making matters simple and efficient, improving things or exploring newer aspects.

Problem solving on computers is the task of expressing the solution of the problem in terms of simple concepts, operations and computer code (program) to obtain the results. To solve problem on computers, we need to:

- define the problem precisely,
- indicate the input data requirements and expected outputs,
- write the various steps to arrive at the solution (setting up an alogirithm)
- prepare a flowchart expressing the solution graphically as this can help to prepare the computer codes quickly and document the solution pictorially as well.
- develop the computer code, that is, program
- test, debug and run the program on the computer to obtain the final results.

It is evident from the above that the problem solving involves the skills to analyze the problem systematically and logically. Now problems may be solved in various ways. It depends on the problem and the kind of procedure used. In this chapter we shall study the process of problem solving on computers and use that in our later study.

2.1 Algorithms

A set of instructions to obtain the solution of a given problem is defined as the Algorithm of that problem. The notion of algorithm, though known since ages, has become very popular for problem solving on computers. Computers need precise and well-defined instructions for finding solutions of problems. If there is any ambiguity, the computer will not yield the right results. It is essential that all the stages of solution of a given problem be specified in detail. correctly and clearly. Moreover, the steps must also be organized rightly so that a unique solution is obtained. Thus, any algorithm must have the following properties:

(1) It should be simple.
(2) It should be clear with no ambiguity.
(3) It should lead to a unique solution of the problem.
(4) It should involve a finite number of steps to arrive at a solution.
(5) It should have the capability to handle some unexpected situations which may arise during the solution of a problem (for example, division by zero).

Algorithms may be set up for any type of problems; mathematical/scientific or business. Normally, algorithms for mathematical and scientific problems involve mathematical formula, but algorithms for business problems are generally descriptive and have little use of formulas. We shall explain the process of setting up of algorithms for both type of problems.

Example 2.1

Design an algorithm to obtain a book on computers from your college library. (Assume that the library is on the fourth floor and your classroom is on the ground floor)

The entire process may be specified as:

Algorithm 2.1

```
(1) Start from the classroom.
(2) Climb the stairs and reach the library.
(3) Search a book on computers.
(4) Have the book issued.
(5) Return to your classroom.
```

Your problem was to have a book on computers. You could solve it by getting it from the library. Steps (1-5) specify the procedure. This is the algorithm. It has been written in a simple and clear way. There is no ambiguity. If Step (2) is written as "Climb the stairs", then it is not clear what to do after having climbed the stairs.

Example 2.2

Design an algorithm to calculate the compound interest, given the principle, rate and time.

If P = Principle, T = Time in years, and R = Rate per cent, then compound interest, I

$$\begin{aligned} I &= \text{Amount} - P \\ &= P(1 + R/100)^T - P \qquad (1) \end{aligned}$$

We will not concern ourselves as to how relation (1) has been obtained, but confine to the procedure of getting the compound interest using this formula. An algorithm may be set-up as follows:

Algorithm 2.2

```
(1) Specify the values of P, R and T.
(2) Use relation (1) to calculate the compound interest.
(3) Write down the calculated values of compound interest.
```

Algorithm 2.2 assumes that time is given in years and rate is also expressed yearly. This may not be so. Suppose, time is t months, but R is yearly. Then before using the formula (1), t time will have to be converted into years. Algorithm 2.2 will now appear as:

Modified Algorithm 2.2

```
(1) Specify the values of P, R and t.
(2) Convert time t (in months) to years as $T = t/12$.
(3) Use relation (1) to calculate the compound interest.
(4) Write down the calculated values of compound interest.
```

Algorithms may be represented diagrammatically by flowcharts. Before we go over to this discussion, we shall study some new concepts which will help to understand the development of algorithms better.

2.2 Data Assignment

A variable is a symbol that may assume different values (data) at different times, but at a given time, it will take only one value. When a variable is given a value, it is said to be initialized or a value has been assigned to the variable. In computer language, the name of variable is designed according to certain rules, and refers to specific computer memory locations where its value is stored.

But here we shall indicate variables by capital English letters to distinguish them from other letters and names. Thus A, B, X, W, V, N, ... or a combination of these (as SUM) will be taken as variable names. Variable A may be assigned a value, say 2, or 16.4 or 112.6 or any other value which you may like. The assignment may be indicated as

$A = 2$

or

$A \leftarrow 2$

or

Store the value 2 in A

When we write

$B = A$ (or $B \leftarrow A$)

it implies: assign the value of "variable A to variable B". If $A = 2$, then B assumes the value 2.

Let

$A = 3.0$, $B = 6.7$, $C = 2.3$, $D = 4.0$

If we write

$$E = A + B + C + D$$

then the value of variable E is

$$= 3 + 6.7 + 2.3 + 4.0$$
$$= 16.0$$

i.e. 16.0
The average of A, B, C, D is

$$P = (A+B+C+D)/4$$

The value of P will be (3.0+6. 7+2. 3+4.0)/4 = 4.0

In Computer Science, the symbol = or ← is referred to as the Assignment symbol or Replacement operator. When a variable is assigned a new value, its old value is lost. This can be understood from the following.

We have seen above that E = 16.0. If we now write

$$E = A$$

then the value of E will be 3.0 (as A is 3.0) and not 16.0. Value of 16.0 has been replaced by 3.0. Now consider

$$\begin{aligned} P = {} & P + B \\ = {} & 4.0 + 6.7 \\ = {} & 10.7 \end{aligned}$$

Here, first the value of P+B is calculated by taking the old value of P i.e. 4.0. The sum P+B = 4.0+6.7 = 10.7. The new value 10.7 becomes the value of P, while its old value 4.0 has been lost.

A variable may appear on the right or left of the assignment operator. The value of variable which is used on the right side is its old value. The new value is obtained after calculating the value of the right hand side. To start with, say N = 0. Then

$$\begin{aligned} N = {} & N+1 \\ & (0+1) \end{aligned}$$

makes N = 1. Next

$$\begin{aligned} N = {} & N + 1 \\ & (1 + 1) \end{aligned}$$

sets N = 2. Further, the statement

$$\begin{aligned} N = {} & N + 1 \\ & (2 + 1) \end{aligned}$$

assigns the value 3 to N. And so on. The variable N is called an Accumulator or Counter. This is a very useful artifice and can be used in several programming applications.

Values can be assigned to variables only and not to the constants as a constant is a value by itself.

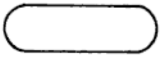
2.3 Flowcharts

A flowchart is a diagramatic representation of the algorithm or of the plan of solution of a problem. It indicates the process of solution, the relevant operations and computations, the point of decision and other information which is a part of the solution. Flowcharts are of particular value for documenting a program. They are constructed by using special geometrical symbols. Each symbol represents an activity. The activity could be input/output of data, computation/processing of data, taking a decision, terminating the solution, etc. The symbols are joined by arrows to obain a complete flowchart.

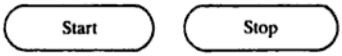
Flowchart Symbols

Flowcharts are designed using standard symbols. There are symbols for various steps of an algorithm. Most commonly used symbols are:

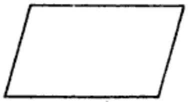
(a) Oval



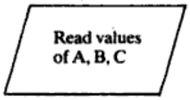
The oval symbol indicates the beginning or the end of a flowchart. Examples of its usage are:



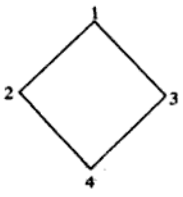
(b) Parallelogram



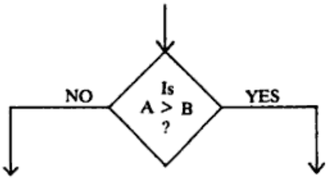
The parallelogram symbol denotes the input/output of information on any device. For example, if we wish to indicate input for variables A, B, C, we may show this with a flowchart as



(c) Decision



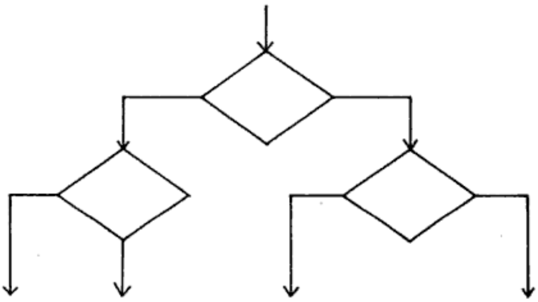
The decision symbol (diamond symbol) indicates the position of making decision in a solution. The entry is indicated at corner 1, the decisions are shown in the box, while the exit may be indicated at corners 2, 3, 4. An example of its use is:

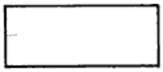


(in actual flowchart, numbers 1,2,3,4 of the corners are not indicated).

The above example implies the evaluation of the condition A > B first. If A is greater than B, then the flow goes along the path indicated by YES. If A is not greater than B, then the flow moves along the path shown by NO

Several of these symbols may also be combined to indicate the paths of multiple decisions as shown below.



(d) Rectangle 

The rectangle symbol indicates some kind of computations in the program. Say, we wish to show the calculations

$$A = B + C + D$$

This can be done as

$$\boxed{\begin{array}{c}\text{Find the sum of}\\ \text{B, C, D and}\\ \text{store in A}\end{array}}$$

or as

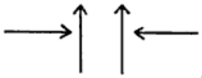
$$\boxed{A = B + C + D}$$

or as

$$\boxed{A \leftarrow B + C + D}$$

An arrow appearing in the flowchart symbol implies assignment. The notation A ← B + C + D indicates: evaluate the sum of B + C + D first and assign it to A. In our flowcharts, we shall adopt this notation.

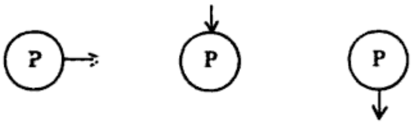
(e) Arrow



The arrow symbol is used to connect the various flowchart symbols and denote the direction of flow of program execution.

(f) Connector

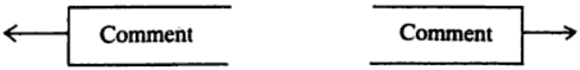
The connector symbols show the connection between the various parts of a flowchart. If in a flowchart the symbols are specified as



they represent the same point.

(g) Comment

The symbol used to indicate comments on the contents of a symbol is



We shall illustrate the use of these symbols to prepare flowcharts and algorithms of various problems.

Example 2.3

Develop a flowchart for the algorithm of example 2.1.
The flowchart may appear as shown in Fig. 2.1

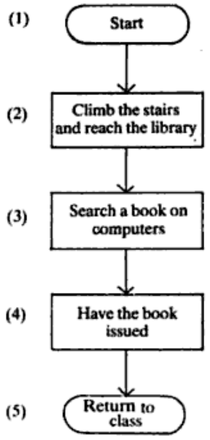

(1)
Start
(2)
Climb the stairs and reach the library
(3)
Search a book on computers
(4)
Have the book issued
(5)
Return to class

Fig. 2.1

It is a convention to use oval symbols at the start and end of flowchart. Numbers of parentheses on left indicate the steps of Algorithm 2.1.

Example 2.4

Develop an algorithm and draw a flowchart to find the average of four numbers stored in variables A, B, C, D.

Algorithm 2.4

(1) Start
(2) Read the values in variables A, B, C, D.
(3) Calculate the average from (A+B+C+D)/4 and store the result in variable P.
(4) Write the number stored in P.
(5) Stop.

A flowchart for this algorithm may be drawn as shown in Fig 2.2. The numbers of brackets on left indicate the steps of Algorithm 2.4 corresponding to each symbol.

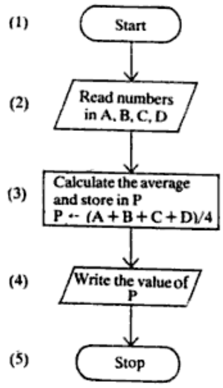
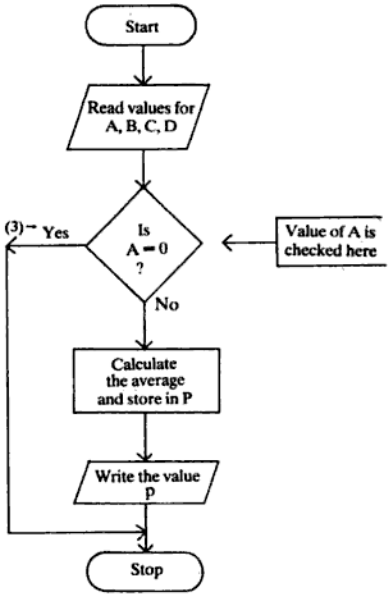


Fig. 2.2



Start
Read values for A, B, C, D
(3)→ Yes
Is A = 0 ?
Value of A is checked here
No
Calculate the average and store in P
Write the value P
Stop

Fig. 2.3

Now, we impose a restriction that when the value of variable A is zero, no averaging is to be done. our algorithm shall take care of this condition. Moreover, this condition should also be shown in the flowchart. Modified algorithm and flowcharts will appear as:

Modified Algorithm 2.4

```
(1) Start.
(2) Read the values in variable A, B, C, D.
(3) Check, if the value of A is zero or not? If A = 0, go to step (6), otherwise
    continue with the next step 4.
(4) Calculate the average of A, B, C, D and store the result in variable P.
(5) Write the value of P.
(6) Stop.
```

The flowchart will be modified as shown in Fig. 2.3
Symbol (3) → indicates the step (3) of the modified algorithm. In Fig. 2.3, value of A is examined in diamond symbol. If the value is zero, the arrow marked Yes

goes to Stop. If the value is non-zero, the arrow marked No takes the flow to the stage of further calculations.

The decision symbol is very useful where three criteria need be examined. We give an example to illustrate this.

Example 2.5

Read a number. Draw a flowchart to examine if the number is positive, zero or negative.

A number is positive, if it is greater than (>) zero. It is negative, if it is less than (<) zero. This criteria can be used to prepare the flowchart. It may be drawn as shown in Fig. 2.4.

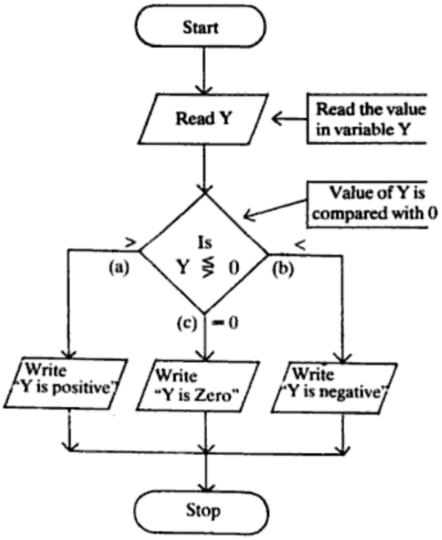


Fig. 2.4

In Fig. 2.4, when tne value of Y > 0 (that is positive), path (a) is followed. When Y <0, path (b) is taken and when Y = 0, path (c) is followed. After either of the paths, (a), (b), (c), the operation Write is performed and then the Stop.

In Section 2.2, you learnt how a variable can be used as an accumulator. Such a variable can act as a counter, that is, it can be used to perform counting. We illustrate this by an example.

Example 2.6

Let there be a set of numbers. It is not known as to how many numbers are there in this set. Develop an algorithm and a flowchart to count them. Write their values.

Algorithm 2.6

(1) Start.
(2) Define a variable M (say) which will act as a counter variable. Initialize it to 0.
(3) Read the number of the given set in variable X (say).
(4) Increase the value of M by 1.
(5) Write the value of X and M.
(6) Go to step 3.

A flowchart corresponding to algorithm 2.6 may be drawn as shown in Fig. 2.5.

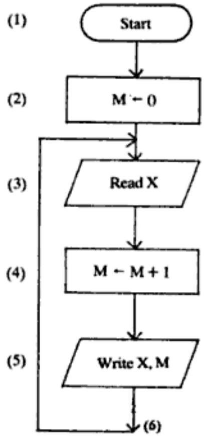


Fig. 2.5

You will see from steps (3) and (4) that when a value of X is read, M is increased by 1. If 13 values of X are read, M will assume the value 13. This is how M acts as a counter.

Let us examine Fig. 2.5 further. It may be seen that there is no termination symbol. More and more data will be required as the control always goes back to step (3). But there must be an end to reading of data. There may be many ways of indicating this. One way of signalling end of data is by the use of a trailer datum. A trailer datum is the last data item in a list. This may be specified by yourself. Suppose it is 9999. (It could have been any other number which is not a part of the given data). Store it in variable XMAX. This number must not be part of the given set of values and should be specified as the last value. This will serve to terminate the reading process. The algorithm of Example 2.6 will become now:

(1) Start.
(2) Introduce a variable M (say) which will act as a counter variable. Set it equal to 0.
(3) Set the variable XMAX to 9999.
(4) Read a number of the given set in variable X (say).
(5) Compare X with XMAX. If X = XMAX, go to step 9, otherwise perform step 6.
(6) Increase the value of M by 1.
(7) Write the value of X and M.
(8) Go to step 4.
(9) Stop.

The flowchart of Fig. 2.5 will be modified and appear as shown in Fig. 2.6.

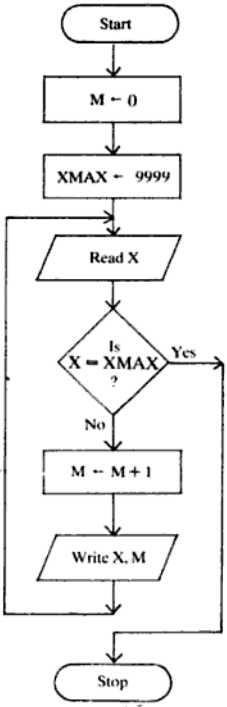


Fig. 2.6

The discussion and various examples of this chapter illustrate how to set up an algorithm of a problem and represent it diagramatically. We shall learn how to solve problems on a computer. In order to do this, it is essential that the algorithm be coded in a language which the computer can 'understand'. The code is called a Program. A flowchart which represents the commands/ statements of a program is called a Program Flowchart.

There are no hard and fast rules about when to use the flowchart. It is usually not worth while to draw the flowchart for small and simple problems; but, for more involved and lengthy problems, flowcharts may save considerable time, effort and trouble. Flowcharts are also of particular aid for documenting a program which may be of general utility and is to be retained and used later on either by the same person who prepared the computer code or some one else.

Remember: both algorithm and flowchart describe the procedure of problem solution, the former is descriptive and notational while the later is its pictorial represenation.

The next stage in problem solving on computers is the development of computer codes for the algorithms/flowcharts and their execution on the computers. This can be done by expressing the algorithm in a computer programming language. In the following section, we shall introduce some of the commonly used programming languages and describe Pascal in brief. Further details of Pascal are going to be the subject matter of the succeeding chapters.

2.4 Programming Languages and Pascal

The algorithm of a problem has to be coded in a language which the computer can 'understand'. The computer needs precise instructions to perform any operation. The instructions to the computer are provided with the help of a programming language by preparing a program. The language whose design is governed by the circuitry and the structure of the machine is known as Machine Language. This language is difficult to learn and use. It is specific to a given computer and is different for different computers. Therefore, general programs that can be run on various machines cannot be written in this language. Such a language is also referred to as Low Level Language. To overcome the difficulties of a machine language, other computer languages have been designed. These are particularly oriented towards describing the procedures for solving the problems and are known as Algorithmic or Procedure oriented or High Level Languages. A large number of high level languages have been developed for specific requirements. They are easy to learn and programs may be written in these languages with much less effort. However, the computer cannot 'understand' them and they need to be translated into the machine language with the help of other programs known as Compilers or Translators.

Programming languages are available which meet the needs of any category of users. Some of the commonly used languages for various applications are:

(a) Scientific and Engineering

BASIC	Beginners All-purpose Symbolic Instruction Code
FORTRAN	FORmula TRANslator
APL	A Programming Langauge

(b) Business

COBOL	COmmon Business Oriented Language
BASIC	

(c) Text Processing

LISP	LISt Processing
SNOBOL	StriNg Oriented symBOlic Language

(d) General Purpose

PL/1	Programming Language One
PASCAL	
ALGOL 68	
ADA	
C	
MODULA-2	

(e) Artificial Intelligence (AI)

PROLOG	PROgramming in LOGic
LISP	

(f) Simulation

SIMULA
SIMSCRIPT

The implementation of a language on a computer depends on the computer system. Some variations of the same language may be found on different systems. The proper source of language implementation is always the manufacturer's manual.

The language Pascal was developed by Wirth in early 1970s. It was designed to serve as a language for teaching computer programming as a systematic discipline and also to develop reliable and efficient programs. Pascal is Algol-based language and has many of the constructs of Algol. In fact, it includes Algol 60 as a subset.

Computer programming languages may be characterized as: strongly typed, typed, weakly typed and untyped or typeless. A language may be said to be strongly typed when the following conditions are met:

(a) the types of all entities appearing in a program are defined prior to their use;
(b) every entity in the program is of unique type;
(c) mixing of types is not allowed;

(d) value type conversion (changing the value from one type to another) is possible only by using special functions.

An example of a strongly typed language is Ada. Languages, where conditions (a) – (d) are not strictly fulfilled, but allow only minor but restricted violations, are generally called typed. An example is Pascal. However, languages, where these conditions are partly violated, may be categorized as weakly typed (an example is Fortran 77) while those languages which do not insist on these conditions are the typeless languages (e.g. Basic).

The advantages of having typing features in a language are discussed later in Chapter 3.

Pascal is a typed language. It offers extensive error checking facilities during compilation and execution phases. Moreover, there are available several data types and a variety of programming structures. All this helps to develop transparent, efficient and reliable programs. Moreover, program development is simplified. It is easy to understand Pascal programs. They are easy to maintain as well. Due to this, Pascal has grown in popularity and has been implemented practically on all computers: personal, micro, mini, mainframe and supercomputers. It is very popular in teaching and academic institutes.

Pascal has been used for Systems Programming as well. Systems programming is concerned with the design, development and production of programs that are required for the preparation, editing, translation, loading, supervision, maintenance, control and running of computers and computer programs. Such programs are called Systems Programs. Examples of such programs are: Operating Systems (O.S.), Compilers, Editors, Assemblers, File Management System, etc. Extended versions of Pascal include features to handle Concurrent Programming as well. (Concurrent Programming refers to the design and development of programs for parallel execution of several processes/tasks. Such programs are used in multisuer O.S., multiprocessing systems, computer networks and computer configurations with separate input/output processors). Several of the features of Pascal have been included in the new programming language Ada.

The Pascal language presented here will be the Standard Pascal version as developed by Wirth and adopted by International Standards Organization (ISO) with slight modifications/alterations.

2.5 Source and Object Programs

A set of instructions of the high level language used to code a problem to find its solution on a computer is referred to as Source Program. The computer translates the source program into the machine language using a compiler. This stage is the Compilation phase. All the testing of the source program as regards the correct format of the instructions, is performed at this stage and errors, if any, are printed. If there is no error, the source program is transformed into the machine language program called Object Program. The object program is executed to perform calculations. This stage is the Execution phase. Data, if required by the program, are supplied now and the results are obtained on the output device. We may indicate this process diagramatically as shown in Fig. 2.7.

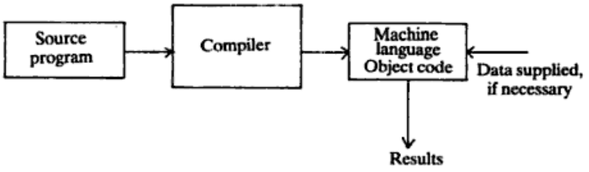


Fig. 2.7

We give below a simple example of a Pascal program which computes the area of a triangle when its three sides A, B, C are given.

Pascal Program Example 1

```
program TRIANGLE (input, output);                    (i)
  var A, B, C, S, AREA : real;                        (ii)
  begin                                               (iii)
    readln (A, B, C);                                 (iv)
    S: = (A+B+C)/2.0;                                 (v)
    AREA: = sqrt (S * (S − A) * (S − B) * (S − C));   (vi)
    writeln (AREA)                                    (vii)
  end.                                                (viii)
```

This is a source program to calculate the area of a triangle. It has 8 instructions numbered (i) to (viii) for reference convenience. The meaning of these lines is explained below.

Line (i) :	Indicates the name of the program as TRIANGLE and specifies that the program will need input data and give results on an output device.
Line (ii) :	Declares variables A, B, C, S and AREA as of real type.
Line (iii) :	Specifies the start of program execution.
Line (iv) :	This instructs the computer to read data values for variables A, B, C.
Line (v) :	Here half the sum of value of variables A, B, C is calculated and stored in variable S.
Line (vi) :	This line causes the calculation of square root of the quantity S(S-A) (S-B) (S-C) and the result is assigned to variable AREA.
Line (vii) :	The value of AREA is written on the output device.
Line (viii) :	It singles the end of the calculations.

Lines (i) to (viii) constitute a Program. Lines (iii) − (viii) form the body of the program while line (ii) constitute the declaration part of the program. Line (i) assigns name to the program.

Pascal programs must have this structure, that is, a program naming/heading statement, declaration and execution part, strictly in this order. We shall discuss more about these in later chapters.

Computer programs are made up of Statements. A statement specifies a step of a program. The step may pertain to declaration, assigning data, reading of data, writing of data, taking logical decisions, transfer of program execution control, and so on. All statements are designed using some specific Pascal word(s). In the illustration Program Example 1, Pascal words have been shown as bold lower case. These are **program, input, output,** (line i); **var, real** (line ii); **begin** (line iii); **readln** (line iv); **writeln** (line vii) and **end** (line viii). Such words are refered to as Reserved words. A complete list of Pascal reserved words is given in Appendix II. A reserved word gives a specific information to the Pascal compiler. It must be spelt and used as defined by the language. Moreover, reserved words are recognized only when they appear at the appropriate place and proper context. They should be used only for the purpose they are defined in the language. Pascal does not allow their redefinition.

In the text, we shall represent all reserved words by bold lower case letters, while the user-defined letters/words will be denoted by capitals. Comments in a Pascal program, enclosed within curly brackets { }, will include both lower and upper case letters. This is the convention pursued by us. However, **Pascal does not put any restriction on the letter/words being lower or upper case. Even mixing of lower and upper case letters is allowed in defining names, etc.**

2.6 Features of Computer Programs

To solve any problem on a computer, its program has to be developed. All computer programs, may be application, system or of any other type, are referred to as Software. (All electronic and mechanical components, forming the computer system, are known as Hardware). The cost of software is increasing day by day while that of hardware is decreasing. Software is becoming expensive due to its enhanced sophistication and increasing cost of human labour and scarcity of trained manpower. Hardware costs have come down because of great advances in electronics technology.

Software systems (collection of computer programs that can cross-reference and interact among themselves) have become very complex. Designing software system is an activity that demands much intellectual capability, logical approach, human efforts and resources. The discipline of planning, designing, creating, managing, maintaining, etc. has come to be known as Software Engineering. It also includes the development of techniques that reduce high software cost, measure software performance, increase reliability and enhance portability, etc. Programming language is a tool for the development of software systems. So any language that is used should be such that the software system has the following main qualities.

(i) *Reliability*

The software system must be reliable, that is, it should work as intended and lead to correct results. Software is correct if it behaves according to specifications.

(ii) *Maintainability*

The software system should be such that it is easy to understand and fix the occurrence of bugs whenever they occur. (Remember that a software system is never bug free). It should be maintainable with reasonable resources and efforts.

(iii) *Modifiability*

Due to high costs of software systems, they cannot be easily dispensed with. Existing software must be easily modifiable and expandable according to the new and varying requirements.

(iv) *Efficiency*

This implies that the software system functions efficiently under the given resources of time and memory. Normally, a software system is said to be efficient if it occupies less memory and takes minimum execution time. If a software system occupies less memory, it may take more execution time. On the other hand, if it is to be executed fast, then it may need more memory. Generally, it is difficult to satisfy these two requirements simultaneously and they need considerable skill and time on the part of the programmer.

The above characteristics of a software system can be achieved by appropriate software development strategies and programming languages. The choice of progrmaming languages becomes very important as this has great impact on the qualities of the programs. The computer programs (constitutents of the software system), that are designed using the language, must have the following essential features.

(a) *Clarity:* implies the easy readability of the program and proper documentation.

(b) *Security:* refers to detection of errors in a program by the computer. Errors may be detected during the compilation phase or the execution phase. Earlier the errors are detected, better it is.

(c) *Transparency:* implies the ease with which we can understand a program and know what it is doing. The more transparent a program, better it is.

(d) *Integrity:* refers to the accuracy of the computations and results.

(e) *Modularity:* implies that the programs be developed using simple units in the form of modules and subprograms.

(f) *Generality:* means that the programs should be of general nature so that they can be used under different situations and requirements.

(g) *Efficiency:* signifies that the program should be executed fast and occupy less memory space.

(h) *Structured design:* implies that programs be designed using some basic constructs of the language only.

(i) *Portability:* refers to running the same program on different computer systems without any or minor changes only.

If a program has the features (a) – (i), then the software system developed using these programs as the components, is expected to meet the requirements of realiability, maintainability, modifiability and efficiency. Some of the properties (a) – (i) may have certain contrary specifications, and balance has to be achieved for the development of a good software system.

Modern programming languages, such as Pascal, Ada, PL/I, C, Modula -2, etc. help to design programs which may have features (a) – (i) to a sufficient measure. Here, our aim will be to learn Pascal and use its features to develop programs and illustrate how the characteristics (a) – (i) appear in the Pascal programs.

We shall develop programs for simple problems to explain principles of Pascal constructs and their use. There are always many ways to write a program for the same problem. We shall be giving only one way of writing the program. Sometimes, the given program may not emphasize features (a) – (i). We will be doing so purposely to high light certain other features of the language.

Structured Programming is a methodology which is very commonly used to develop programs and software systems. We shall attempt to follow this technique as far as possible. For the present, you need not bother about it. However, after having learnt control and iterative statements, you read Chapter 15 and understand Structured Programming.

2.7 Syntax Diagrams

The syntax of all the symbols, characters, and other entities, such as constants, variables, statements, functions, and so on, occurring in Pascal language can be described by Syntax Charts or Diagrams. They indicate completely the syntactic specification of Pascal entities. Syntax diagrams are useful because they give pictorial representation of the language constructs. We shall explain briefly the design of syntax charts here so that they can be introduced in our discussion later.

Every language, may be English, Hindi, Punjabi, French, etc. has its own syntax. Syntax of a language is a set of rules by which various entities occuring in that language are defined. For example, take the case of English. There are words, sentences, paragraphs, etc. Words are made up of letters, sentences are made up of words, punctuation marks. Similarly, paragraphs are designed by sentences. Syntax diagrams may be used to show the design of words, sentences and paragraphs. We demonstrate how these entities may be represented by syntax diagrams and then go over to the syntax diagrams for Pascal.

English has its own alphabet consisting of letters A-Z, a-z, punctuation marks. For convenience, we consider only the small letters and commonly used punctuation marks , ; ! . ? - . We make the following convention for representation:

(i) Every occurrence of the basic symbol, such as a letter, which cannot be defined in terms of other quantities/symbols, is represented in the syntax diagram by an oval symbol.

(ii) Punctuation mark is represnted by enclosing it a circle.

(iii) The entities that are further defined by other diagrams are represented by rectangles.

Thus, we use only three symbols shown in Fig 2.8.

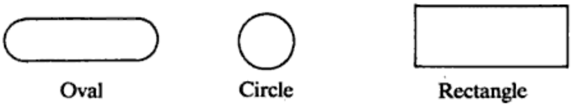


Fig. 2.8: Syntax diagram symbols

to prepare the syntax diagrams. These symbols are connected by arrows.

(iv) the repetition or recursion is represented in the chart by the graph that traces back to itself. An illustration is given in Fig. 2.9(a).

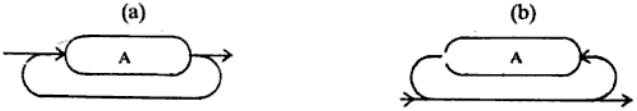


Fig. 2.9: Repetition and recursion in syntax charts

Diagram 2.9(a) implies that the repetition of A must take place at least once. For the case that the repetition may or may not take place at all, Fig. 2.9(b) should be used.

We apply the conventions (i) – (iv) to represent entities in English language.

⇒ You know that the letters i, o, u, a, e are the vowels. We say that a vowel may be i| o| u| a| e|. (The symbol | implies OR). These are the basic symbols so we can represent a vowel by a syntax chart as shown in Fig. 2.10.

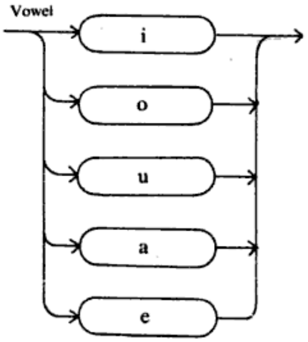


Fig. 2.10: Syntax chart of a vowel

This chart indicates that a vowel may be i or o or u or a or e.

⇒ A word consists of repeated letters, so a word may be represented as shown in Fig. 2.11.

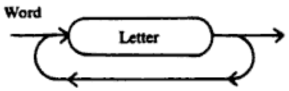


Fig. 2.11: Syntax chart of a word

⇒ An English sentence consists of words and punctuation marks. Consecutive words are separated by spaces. For simplicity, we assume that blank space (denote it by ƀ) is also a punctuation symbol. The syntax diagram of punctuation symbols may be drawn as showin in Fig. 2.12.

Punctuation mark

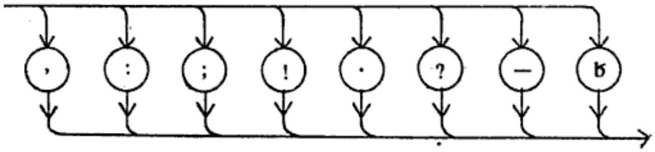


Fig. 2.12: Syntax chart of a punctuation mark

⇒ The syntax chart of a sentence appears as (Fig. 2.13).

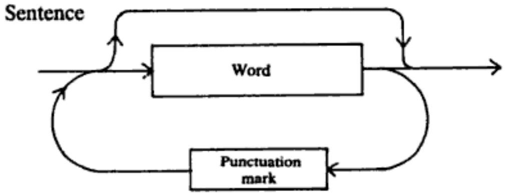


Fig. 2.13: Syntax diagram of an English sentence

A paragraph is a repeated collection of sentences, so its syntax chart is straightward.

Thus, we observe that the structure of the common English language entities and constructs can be easily described in terms of the syntax diagrams.

The syntax diagrams of Pascal language entities and constructs can also be prepared similarly. In order to do so, let us see what constitutes Pascal programs. Every Pascal program consists of

- reserved words
- operators and separators
- user-defined entities

These entities are combined or used according to well-defined rules of Pascal language. These are the Syntax rules. (You will learn these while studying Pascal language and program writing). All Pascal constructs are designed using the above entities. In order to prepare syntax diagrams, we introduce the following notation :

(a) Oval : to represent the Pascal reserved words
(b) Circle: to denote the characters of the Pascal character set (Chapter 3).
(c) Rectangle: to represent a syntactic entity which can be defined by another syntax diagram.

The notation, introduced above, is similar to what we had for English language, except that now entities of Pascal language are to be used to design the syntax charts.

We shall illustrate the design of syntax diagrams for a few entities here and defer the description of others.

*Digits may be 0 | 1 | 2 | 3 | 4 | . . . | 9 in Pascal. Its syntax diagram is as shown in Fig. 2.14.

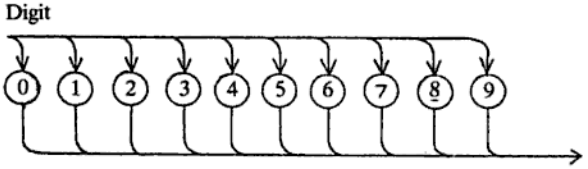


Fig. 2.14: Syntax chart of a digit

* The syntax chart of an identifier appears as given in Fig. 2.15.

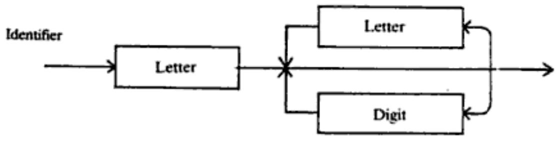


Fig. 2.15: Syntax diagram of an identifier

* The syntax diagram of a Pascal program is as shown in Fig. 2.16.

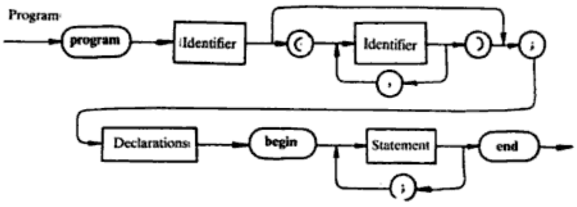


Fig. 2.16: Syntax diagram of Pascal Program

We shall describe the Pascal syntax diagrams further as we go along. As you will note that such diagrams are a powerful tool for defining the language constructs and other entities. The complete syntax diagrams of Pascal are also given in Appendix V for ready reference.

Syntax diagrams help us to give the syntactic specifications of the language entities and constructs. There is another way in which the syntactic specifications may be described. This is what is known as the Backus–Naur Form (BNF). BNF was originally developed by Backus and Peter Naur for the specification of the syntax of Algol 60. This method of specification has been widely used in the field of Computer Science since then. The various entitites and constructs of Pascal can also be expressed in the BNF. We have given these in Appendix IV. The reader can understand it easily by going through it carefully.

The syntax diagrams and BNF descriptions of language syntax are equivalent—the former is pictorial while the later is notational and descriptive representation. The reader is urged to understand both of these.

In this chapter, we have explained the process of problem solving on computers and other elementary concepts. The last step is the development of computer programs and their execution. You have to learn Pascal and write programs in this language. This is what is going to be covered in the subsequent chapters.

Exercises 2

2.1. Complete the following sentences :

(a) A variable is a symbol which may assume different .... at different times.
(b) A flowchart is a .... representation of the ....
(c) The most common symbols used to design flowcharts are ....
(d) The computer needs precise .... to perform any task.
(e) The language, whose design is governed by the circuitry and structure of the machine, is known as .... or .... language.
(f) High level languages are also known as .... or ....
(g) Pascal is a .... language.
(h) Systems programming is concerned with ....

(i) Concurrent programming refers to the design and development of .... for .... execution of several processes/tasks.
(j) The computer translates the .... program into .... language using a ....
(k) Programming is a tool to develop .... systems.
(l) The discipline concerning planning, designing, creating, managing, maintaining, measuring software reliability and performance is known as ....
(m) The important qualities of any software system should be ...., ..., ..., and ....

2.2. Tick the correct words in the following sentences :

(a) The value of a constant/variable does not change in a program.
(b) The symbol used to indicate the start of a flow diagram is oval/rectangle.
(c) A algorithm/flowchart can be used to represent pictorially the procedure to solve a problem.
(d) High level language programs are generally portable/unportable.
(e) Pascal offers good/poor error checking facilities during compilation and execution phases.
(f) External data are supplied to any program (if needed) during compilation/execution phase.
(g) Pascal program has a pre/un-defined structure.
(h) A software system is said to be efficient if it occupies less/more memory and needs large/small execution time.
(i) Accuracy of the computations and results is referred to integrity/security of a program.
(j) Structured/modular design implies developing programs using only some basic constructs of the language.
(k) Clarity/transparency implies easy readability and proper documentation of the program.
(l) The syntactic specification of Pascal entities can be described by flow/syntax charts.
(m) The syntax diagrams and BNF descriptions of language syntax are equivalent/different.

2.3. What do you understand by Problem Solving on Computers? Explain the various steps involved in this process.

2.2. Define an algorithm of a problem. What should be its characteristics?

2.5. Develop an algorithm to host your birthday party at your residence.

2.6. Design an algorithm to make a long-distance telephone call to your girl/boy friend.

2.7. What do you understand by 'data assignment to a variable'? Assign 7.8, 13.13, 113.13, − 2.3, 1313 to variables A, B, C, D, E respectively.

2.8. Let P = 2.13, Q = − 16.23, T = 1.806. Consider

$R \leftarrow P+Q+T$
$P \leftarrow P+T$
$T \leftarrow P+R$

What will be the new values of P, R, T?

2.9 How can a variable be used as a counter? Illustrate by an example. What is the utility of such a facility?

2.10. Draw the basic flowchart symbols. Explain the need and use of flowcharts.

2.11. Prepare flowcharts for exercises 2.5 and 2.6.

2.12. The roots of a quadratic equation

$$a x^2 + b x + c = 0$$

are given as

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Here a, b, c are parameters whose values are given. Develop an algorithm and a flowchart to calculate the real roots of the quadratic equation.

2.13. Develop an algorithm and a flowchart to find the L.C.M. and H.C.F. of two positive integer numbers.

2.14. Prepare an algorithm to prove that a parallelogram is a square if and only if its diagonals are equal and are at right angles to each other. Draw the flowchart of your algorithm.

2.15. Who designed Pascal? What were the objectives behind its design? Mention its main areas of applications.

2.16. Explain
- Source Program
- Object Program
- Compilation phase
- Execution phase
- Reserved Word
- Systems Programming
- Concurrent Programming
- BNF

2.17. What is a Software System? Describe the salient characteristics which any software system must have.

2.18. Explain the important features that every program should have.

2.19. Bring out the features of strongly typed languages. Give examples of strongly typed, typed, weakly typed and typeless programming languages. Give the need of having Type features in a language.

2.20. What is a syntax diagram? Explain its singificance by giving examples.

2.21. What is a program? Give the structure of a Pascal program and draw its syntax chart.

2.22. Prepare syntax charts for the following:

(a) assume that a letter can be any of the capital letters A-Z or small letters a-z,
(b) an English language paragraph,
(c) colour which can be violet/indigo/blue/green/yellow/orange/red,
(d) binary digit,
(e) hexadecimal digit,
(f) an unsigned integer number.

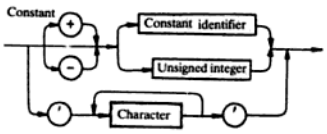
2.23. Redraw the syntax diagram of a sentence using the symbols oval and circles only. Study this diagram carefully and enumerate the type of sentences that can be designed with this chart. Is there any type of English sentence that cannot be generated from this diagram?

2.24. Study the BNF representation of Pascal given in Appendix V. Use it to describe

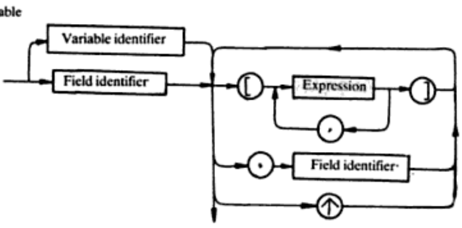
(a) digit (b) letter (c) constant
(d) variable (e) vowel (f) punctuation symbols
(g) English sentence

2.25. Explain the meaning of the following Pascal syntax charts :

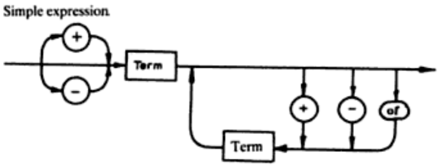
(a)



(b)



(c)



2.26. Discuss the differences between syntax charts and flowcharts.

Chapter 3

Elementary Concepts and Primitive Data Types

We have seen that a computer program can consist of many quantities (entities)- constants, variables, functions, operators, and so on. Each quantity has a specific meaning and can be used in a program according to pre-defined rules of the language. Morever, all the quantities are designed according to certain syntax rules. We shall introduce the basic quantities such as character set, numbers, identifiers, data types, etc. and illustrate their use in the design of expressions, statements and programs as we go along.

3.1 Character Set

Pascal is a written language. Like any other language, it has its own character set or alphabet. All the quantities defined in Pascal are constructed using the characters from this alphabet. The character set is

Letters :	A-Z (upper case)
	a-z (lower case)
Digits :	0-9
Special symbols :	+ − * / := . , ; .. ( ) [ ] ≤ < = > ≥
	<> { } ' blank (We shall indicate a blank space by the symbol
	ƀ in the text)

The characters do not have any significance by themselves. They are assigned meaning when used in an appropriate defined context according to the rules of Pascal language. As for example, the symbol / (slash) is interpreted as division operator when it has operands preceding and following it as 13.13/5.0.

3.2 Numbers

A number is defined by using the digits and the decimal point (.). A number may be positive or negative. A negative number is indicated by specifying the minus sign (−) before it. Use of plus sign (+) before a positive number is optional. No other special symbol is permitted to be used in a number.

Numbers may be of two kinds: integer and real.

Integer Number

An integer number is a number without a decimal point. It is a whole number, no fractional part. Examples of integers numbers are

Valid

7.
−13
27
+12493
0

Invalid	*Reason*
4.81	contains the decimal point
345,691	has the special character comma (,)

Real Number

A ***real number*** is a number with a decimal point. At least one digit must precede and succeed the decimal point. Thus, the decimal point should never appear at the start or end of the number. Examples of real numbers are:

Valid

0.2
−11.346
+23.45
123.0
−0.00067

Invalid	*Reason*
456	decimal point missing
85.	decimal at the end is not followed by at least one digit
4,326.3	special character comma (,) not allowed
3b98.0	blank (b) in a number not permitted
.13	decimal at the start should have at least one digit before it.

A ***real number*** can also be written in another form, called Exponent form. In this form, a real number consists of two parts : a decimal part and an exponent part. Consider

$$\begin{aligned} 23.456 &= 23.456 \times 10^0 \\ &= 2.3456 \times 10^1 \\ &= 0.2345 \times 10^2 \\ &= \underset{\substack{\uparrow \\ \text{decimal part}}}{2345.6} \times \underset{\substack{\uparrow \\ \text{exponent part}}}{10^{-2}} \end{aligned}$$

and so on

Here 23.456 is a number in the decimal form, while all other representations are in exponent form. The exponent part is a scale factor expressed as an integral power of 10. The power may be negative or positive. The part preceding exponent part, that is decimal part, is called the Mantissa. The mantissa may or may not carry the decimal point. When not specified, the decimal is assumed to be after the last digit. Symbol E (or e) is used to denote the base 10 of the exponent part. Illustrations of exponent form of real numbers are:

Valid

23.7E−8
34.89E+12
13E7
−0.49e+13

Invalid	*Reason*
7. E−5	decimal at the end of the mantissa not allowed by a digit
E 9	mantissa missing
1.8E	digits following E missing
43,123 E−11	mantissa contains comma
−6.93e4.0	exponent must be an integer
16ƀ34 E−25	blank in mantissa not allowed

When the decimal in the mantissa appears at the leftomost position, the number is said to be in a Normalized Exponent form. Examples are: 0.123E+6, 0.1934E−10, −0.627E+27, and so on. Real numbers in the computer memory are stored in the normalized form as shown in Fig. 3.1.

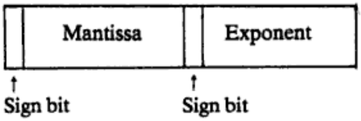


Fig. 3.1: Storing numbers in the normalized form

Mantissa and exponent part may have any sign, + or −. Their signs are indicated in the computer storage according to the system of numeric data representation in the computer memory.

3.3 Identifiers

Various entities such as constants, variables, types, functions, procedures, records etc. occuring in a Pascal program can be assigned names. The names serve to identify them. The technical word for name is Identifier.

An identifier is a sequence of letters and digits which must always begin with a letter. Special symbols and blanks are not allowed to appear in an identifier. An identifier may be as long as you wish, but actual Pascal implementations place restriction on the length of the identifier. Normally, the first 8 characters of an identifier are recognized by most Pascal compilers. Thus, identifiers denoting different entities in a program must differ in their first 8 characters. (In the text, we shall use identifiers consisting of any number of characters). Moreover, there is no restriction on whether to use upper-case or lower-case letters while writing identifiers. In fact, even mixing of lower and upper-case letters is allowed. This is a very useful facility available in Pascal and should be followed as it increases the readability of the Pascal programs. For instance, the identifier RateofInterest is more readable than either RATEOFINTEREST or rateof interest. (However, in this book, we have indicated the user-defined identifiers by capitals for convenience of explanation in the text and to avoid the confusion which is likely to occur due to mixing of lower case words/names with other matter.)

The syntax diagram of an identifier appears as down in Fig. 3.2.

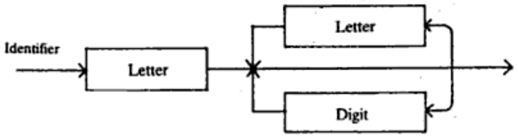


Fig. 3.2: Syntax chart of an identifier

Reserved words are not allowed to be used as identifiers (Appendix II).

Examples of valid and invalid identifiers are:

Valid

A22
Velocity
ROOT2
PAYROLL
TRIANGLE
INCOMTAX

Invalid	*Reason*
3DOWN	begins with a digit, the first character must be a letter
LIGHT-SPEED	contains special symbol
BEGIN	reserved word
AREA.4	decimal not permitted

Identifiers may be assigned certain attributes, such as integer, real, boolean, etc. This is done by declaring them of particular type.

3.4 Data Types

You have seen that numbers may be integer or real. Numbers, such as 6, 61,

−89, 1313,. . . . . . are integers and are referred to as of type integer. Similarly, numbers 6.6, 71.29, −111.3, 0.39999 e+11, ... are real and are said to be of type real. Various quantities such as constants, variables, etc. occuring in a Pascal program must have a type associated with them. There can be one and only one type associated with an entity. The type of an entity establishes the following information about it:

- its meaning
- constraints applicable to it
- possible values that the entity can assume
- operations that can be performed with/on the entity
- functions that can be used with it
- mode of storage in the computer memory

Say, for example, PART is an identifier of type integer; then it can take only integer numbers as its values and that too within a certain defined range of numbers, depending on the Pascal compiler. Suppose the range is from −9999999 to +9999999. If PART is assigned any other value, it will be treated as a mistake.

Other examples of data types are boolean and character. Integer, real, boolean and character types are referred to as Standard or Primitive data types. They are automatically provided as part of the Pascal langauge. The user can also define his own data types as per the requirements of his program and the specific needs of the problem. Examples being subrange and enumerated data types (Chapter 6). This is a very important facility available in Pascal.

Data types may be characterized as Scalar and Structured type. Examples of scalar data types are integer, real, boolean, character, subrange and enumerated (Chapter 6). Structured data types are formed from scalar types. Arrays, records, files and sets are examples of such data types. Pascal also supports another data type called Pointer data type. All the data types available in Pascal are summarized in Fig. 3.3. We shall confine our discussion to scalar data types here and defer the discussion of structured and pointer data types to later chapters.

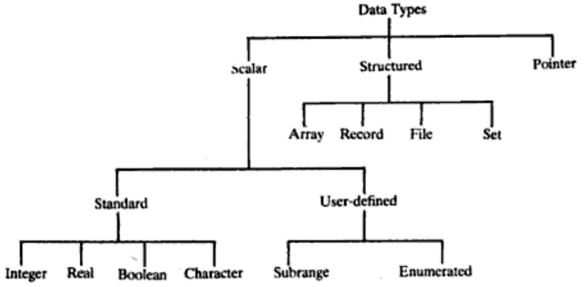


Fig. 3.3: Various data types available in Pascal

The existence of a large number of data types is quite useful. The user can choose the right kind of data type or define a new one according to his requirements. Pascal compiler imposes certain rules and restrictions on the use and specification of data types. Due to this, Pascal is also called a 'typed' language. Major advantages of having the data type facility in a language are:

* Once the data type of an entity is defined, the compiler is automatically informed of the possible attributes, values, and operations that are permitted on that entity.
* The data types enforce discipline and consistency of use on the part of the programmer.
* Data types provide protection from certain programming errors.
* Inconsistent use of data types is automatically pointed by the compiler.

These concepts will get clarified as we go along and learn more about the use of data types.

3.5 Type Declarations

The type of an identifier must be declared. This can be done with the **type** declaration.

```
type
   type-identifier-1, type-identifier-2, . . . . . = type-specifier
```

or as

```
type
   type-identifier-1 = type-specifier-1 ;
   type-identifier-2 = type-specifier-2 ;
        ......................
        ......................
        ......................
```

where

type-identifier	user-defined identifer which can be used to define the type of other entities.
type-specifier	may be **integer, real, boolean, char**, user-defined or other permitted Pascal type.

The sign of equality (=) separates the *type-identifier* and the *type-specifier.*

Let us first study the standard data types.

(a) The type **integer**

The type **integer** allows to represent, store and manipulate integer entities. We can define constants, variables, functions or expressions of integer type. Pascal defines the various operations and operators allowed with integer type of quantities. For example, the following arithmetic operators can be used with integer type entities: *, + , −, **div** and **mod** (see Chapter 4 for more details).

An integer number may vary from − ∞ as + ∞, but, you know that the maximum number that can be stored in a computer memory, depends on the word size. For example, a 16-bit word can store a maximum integer constant as 32767 while a 32-bit word can store $2^{31}-1=2147483647$. Pascal defines an implementation-dependent standard identifier **maxint.** The integer type consists of all the values in the range:

−maxint to **+maxint**

Suppose we wish to define DAYS, EXPENSES of type integer. This can be done as follows:

```
type
     DAYS, EXPENSES = integer ;
```

(b) The type **real**

The type **real** allows to define, store and manipulate real entities, such as constants, variables, functions, expressions, etc. Different opeators are also defined for them. For example, the allowed arithmetic operators are: *, / , + , and −, (for further discussions, see Chapter 4).

Suppose, we wish to declare identifier MONTH, AREA as of type real. This can be done as:

```
type
     MONTH, AREA = real ;
```

(c) The type **boolean**

The type **boolean** enables us to define, store and manipulate logical entities, such as constants, variables, functions and expressions, etc. The operators defined for such type of data are: **and, or, not.**

Say, we want to define identifiers IC, YES and TRUTHVALUE as of type boolean. This can be done as;

```
type
     IC, YES, TRUTHFALUE = boolean ;
```

(d) The type **char**

The type **char** (character) allows us to define and manipute character quantities: constant, variables, functions, etc.

Suppose it is desired to declare BOOK, AUTHOR as identifiers of type character. It can be done as follows:

```
type
     BOOK, AUTHOR = char ;
```

The type specification of the identifier only defines its type and nothing else. The semicolon used after the type specifier signals the end of declaration and acts as a separator. There may be several and different declarations on the same line. As for example, the following declarations

```
type
    A, B, C : integer ; P, Q : real ; BB : boolean ;
```

on a single line are allowed.

3.6 Constants

A constant is an entity which remains unchanged during the execution of a program. Pascal allows the following kind of constants as shown in Fig. 3.4:

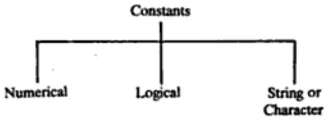


Fig. 3.4: Type of constants

Numerical Constants

Numbers, integers or reals, are examples of numerical consants. Such constants are used in arithmetic expressions and can be assigned to numeric variables only.

Logical Constants

There are only two logical constants: **true** and **false**. These can be assigned to the logical variables only.

Character Constants

A sequence of characters is called a String. The characters may be numeric, letters, blank, special characters or a combination of these. Examples of strings are : A1B/C, PAYINGƀGUEST, RAM and so on. When a string is enclosed within quotes, it is called a String or Character Constant. Examples of such constants are:

'A1B/C'
'PAYINGƀGUEST'

If a quote appears in a string, such as

'MY FRIEND'S FATHER'

then an additional quote is inserted with the existing quote and the entire string is further enclosed between quotes as

'MY FRIEND''S FATHER'

Constants may be assigned names as well. This is done in the declaration part of the program by specifying the declaration **const**. Its format is

---

const
identifier = *constant*

---

where

identifier	name by which the constant will be known in the program
constant	constant value
=	the equality sign which assigns the value to the *identifier*

Examples of constant declarations are

```
const
   LIGHTVELOCITY = 3.0E + 10 ;          (real)
             PIE = 3.14159 ;            (real)
          HEIGHT = 345 ;                (integer)
      AUTHORNAME = 'NANAK SINGH' ;      (string)
        OPERATOR = '+';                 (string)
LOGICALCONSTANT = true ;          (boolean)
```

The type of the identifer is the same as that of the constant as indicated on the right. In the same declaration, the following type of specification is not allowed:

```
const
  X = 20.13 ;
  Y = X
```

Moreover, a name cannot be assigned to an expression consisting of constants, such as

```
const
  A = 3 * 13
```

or a variation of this.

All the constant declarations must be given before variable declaration in a Pascal program.

The use of *constant* identifiers normally makes a program more readable. It also helps in documentation of the program. Moreover, it enables the programmer to club together special quantities (may be machine dependent or example dependent) at one place at the beginning of the program where they can be easily modified, added or deleted. This helps to make programs portable.

3.7 Variables

A variable is an identifier which always refers to the computer memory space where some datum can be stored. This datum is said to be the value of the variable. A variable can assume different values in a program at different stages of the program execution.

All variable identifiers appearing in a program must be declared before their use. The syntax of a variable declaration is:

```
var
  list-1 : type-specifier-1;
  list-2 : type-specifier-2;
  list-3 : type-specifier-3;
      :          :
      :          :
```

where

var	keyword which indicates that the identifiers following it are variables,
list-1, *list*-2.....	variable names separated from each other by comma,
type-specifier	type of identifiers; may be **integer, real, boolean, char**, user- defined or some other allowed type.

list and *type-specifier* must always be separated from each other by a colon (:). The semicolon separates the different declarations.
Examples of variable declarations are

```
var
  DAYSOFMONTH          : integer ;
              X,Y,Z    : real ;
        BOX, BYTE      : boolean ;
  MYNAME, YOURNAME     : char
```

Here DAYSOFMONTH is an integer variable; X, Y, Z are of type real; BOX, BYTE are boolean variables while MYNAME and YOURNAME are character variables.

A value assigned to a variable should be of the same type as the type of the variable, that is, *type of value and variable should match*, otherwise Pascal compiler will give an error. However, there is one exception to this rule. Real and integer type may be mixed under certain conditions as we shall see later.

A value may be assigned to a variable by an Assignment statement using the assignment operator := as (see Chapter 5 for more details).

```
DAYSOFMONTH := 30 ;
          X := 10.3 ;
       BYTE := true ;
   YOURNAME := 'MONA'
```

Thus, a character variable is assigned a string constant while a boolean variable must be initialized to **true** or **false** values. In the case of numeric variables, there is a little flexibility which is: a real variable may be assigned a real number or an integer number as its value. When the value is integer, then it is converted into the real form and stored inside the computer memory at a place specified by the variable name. It may be mentioned here that there is a distinction between an integer number of type real and an integer number of type integer. For example, 5 is integer while 5.0 is real, though integer in value. The numbers 5 and 5.0 have different representations in the computer memory.

We have seen that an identifier may be assigned a type or in other words, a *type* can be identified by an identifier. This *type* identifier can be used to define variables of that *type*. This is as follows:

```
type
   id-1 = t-1 ;
   id-2 = t-2 ;
   :      :
   :      :

var
   a, b, c, . . . . .: id- 1 ;
   p, q, . . . . . : id-2 ;
   :              :
   :              :
```

Here identifier *id*-1 is of type *t*-1 while *id*-2 is of type *t*-2 and so on. Next, the variables *a, b, c* . . . . . have been declared to be of type *id*-1, that is *t*-1, whereas variables *p, q* . . . . . have been declared of type *id*-2, that is, *t*-2.

Types *t*-1, *t*-2 . . . . may be standard or user-defined. The above procedure is an alterantive way of defining **type** of variables.

We have seen that Pascal allows the declaration of type, constants and variables separately in a program. These declarations must be specified in the order as:

```
const    declaration
type     declaration
var      declaration
```

otherwise error will occur.

The reader should appreciate the differences between **type** declaration and **var** (variable) declaration. With type specification, we can indicate the category/class of identifiers (such as integer, real etc.) whereas the variable specification implies the type of values which the variable name can assume. Moreover, the variable identifier refers to the memory locations where the value of the variable is going to be stored. There is no such thing associated with the **type** declaration.

3.8 Standard Built-in Functions

Several functions are supplied as part of the Pascal system. They are called Built-in or Library functions. Such functions have pre-defined names. Each has an argument which is always enclosed within parentheses. Functions are available which operate on quantities of different types. Let us study them separately.

(a) *Arithmetic built-in functions*

(i) Functions for the real data type which produce real results (x is the argument of the function which should be real)

• **abs** (x)	$\vert x \vert$	
• **sqr** (x)	x^2	
• **sin** (x)	sine of x	x
• **cos** (x)	cosine of x	in
• **arctan** (x)	$\tan^{-1}(x)$	radians
• **ln** (x)	natural log of x	$(x > 0)$
• **exp** (x)	e^x	
• **sqrt**	$\sqrt{x}$	$(x \geqslant 0)$

(ii) Functions that produce integer results (y is argument of the function which should be integer)

• **abs** (y)	$\vert y \vert$
• **sqr** (y)	y^2

- **trunc** (x) x is real; **trunc** (x) returns intger part of x as its value while decimal part is dropped; for example, **trunc** (6.3) ⇒ 6
- **round** (x) x is real; **round** (x) rounds the value of x to the nearest integer; for example,
  round (3.6) ⇒ 4
  round (−3.4) ⇒ −3

In fact, **round** (x) gives the same value as **trunc** (x + 0.5). This is true for both positive and negative values of x.

(b) *Boolean built-in functions*

- **odd** (y) returns the value **true** if integer y is odd, otherwise value is false
- **eoln** (F)† returns the value **true** if the end of a line in a file F has been reached, otherwise value **false** is returned.
- **eof** (F)† returns the value **true** if the end of the file F has been reached, otherwise value **false** is returned

†The reader may skip these functions for the present.

(c) *Character built-in functions*

The characters in the character set are ordered according to a predefined sequence during the Pascal implementation. This is referred to as the Collating Sequence. This sequence may vary from computer to computer system. For illustration purposes, we assume the following sequence:

- the lower case letters have the sequence : a, b, ... z such that a < b < c < ...... < z.
- the digits follow the natural sequence: 0, 1, 2, ... 9, such that 0 < 1 < 2 <.... < 9.
- the set of digits follows the set of letters.

Thus, we assume that the letters and digits have the sequence

a, b, c, ..., z, 0, 1, 2, 3, ..., 9

This enables us to define a one-to-one correspondence between the above sequence and a set of integers as

a	b	c	d	....z	0	1	2	3	9	
0	1	2	3	26	27	28	29		35	← ordinal values

Thus, each character has an integer number associated with it. This number is called the Ordinal number or value. All characters in Pascal set have a unique ordinal value which is implementation dependent. To obtain the precise ordinal value of a given character, reference to the installation manual should be made. However, for our discussion, we shall assume the above ordinal values. Moreover, boolean constants **false** and true are also ordinal. **false** precedes **true.**

Pascal defines functions which enable to find the ordinal numbers of characters in a character set or the reverse of this. It is also possible to locate the succeeding and preceding characters/numbers in a set. Such functions are discussed below:

- **ord** ('*character*') takes a *character* argument and returns an integer value which represents the ordinal value of that *character.*

Examples are

ord ('a') = 0
ord ('d') = 3
ord ('9') = 35

- **chr** (*J*) takes the integer ordinal number *J* as its value and returns the character corresponding to that ordinal value.

Examples are

chr (3) = 'd'
chr (35) = '9'
chr (6) = 'g'

In general, if C is a character, then

$$\textbf{chr}\ (\textbf{ord}\ (C)) = C$$

If J is an integer in the range of ordinal values corresponding to the character set, then

$$\textbf{ord}\ (\textbf{chr}\ (J)) = J$$

We have seen that integer, real, boolean, character or enumerated-type can be of scalar or structured type. Further, we can associate ordinal numbers with integer, boolean, character and enumerated data types only. Due to this, these are also known as Ordinal Data types. Why can't we associate an ordinal number with real data type (?). Each value of a given ordinal type has a unique predecesser and a unique successor. This can be obtained by the use of two built-in functions. These are

- the predecessor function

    pred (x)

where x the argument whose type may be an integer, character or boolean. The function **pred** (x) returns the value preceding x.

Example of **pred** function are:

pred ('b') = 'a'
pred (5) = 4

- the successor function

    succ (x)

where x the argument whose type may be an integer, char or boolean; the function **succ** (x) returns the character succeeding character indicated by the argument x. Example are

succ ('b') = 'c'
succ (8) = 9

The following relationships exist between the character functions

pred (x) = **chr** (**ord** (x) − 1)
succ (x) = **chr** (**ord** (x) + 1)

The function **pred** and **succ** can have integer and boolean arguments as well. In the case of boolean constants, **true** succeeds **false.** As for example.

pred (10) = 9
succ (25) = 26
pred (**true**) = **false**
pred (**false**) = not defined

The actions of these functions are summarized below:

Function	*Identifier*	*Argument type*	*Result type*
ordinal	**ord**	character	integer
character	**chr**	integer	character
predecessor	**pred**	character/integer	character/integer
successor	**succ**	character/integer	character/integer

The names of built-in functions are reserved words and should not be used as identifiers. While developing programs, built-in functions should be employed wherever required. They save coding effort and computing time as they have been developed by experts and written in an efficient way.

In this chapter, we have considered the standard data types as defined in Pascal language and are automatically available to the user. As mentioned earlier, the user can define his own data types as well. Most of the built-in functions are applicable to the user-defined data types also. We shall defer the discussion of such data types to Chapter 6. Next, we go over to describe the use of constants, variables and standard functions to design expressions and their evaluation.

Exercises 3

3.1. Tick the correct answers in the following lines:

(i) An integer number is a number with/without a decimal.
(ii) The decimal point must/must not appear at the start or end of the number.
(iii) In a normalized form of the real number, the decimal point is at the extreme right/left position of the number.
(iv) Reserved words can/cannot be used as identifers.
(v) Data type is a good/bad facility available in Pascal.
(vi) Constants can/cannot be assigned names in Pascal.
(vii) String constants are same/different from the character constants.
(viii) Type of each variable must/need not be given in a Pascal program.
(ix) The operator := and = are same/different.
(x) Characters in the character set are implemented on a computer system in a ordered/random way.

3.2. Answer as true or false:

(a) Pascal does not allow the use of lower case letters. (T/F)
(b) The ordinal numbers of the boolean constants **true** and **false** are identical.(T/F)
(c) Extremely small numbers cannot be written in the exponent form. (T/F)
(d) Blanks are allowed in an identifier.(T/F)
(e) Pascal does not allow the use of any other data type except scalar. (T/F)
(f) Constant declaration may follow the variable declaration (T/F)
(g) All characters in Pascal character set have been assigned ordinal values. (T/F)
(h) Type of a variable must be defined before its use. (T/F)
(i) The built-in functions are applicable to the standard data types only. (T/F)
(j) Pascal does not allow the user to define his own data types. (T/F)

3.3. Explain the significance of character set of a language. Compare the character set of Pascal (as given in the text) with the character set as available on your system. List all the characters of your terminal which are not part of the standard alphabet.

3.4. What is a number? Prepare its syntax diagram. Explain the difference between integer and real numbers. Illustrate by examples.

3.5. Describe the advantages of representing real numbers in the exponent form. How are such numbers stored in the computer memory?

3.6. Separate the valid and invalid numbers form the following list. Write the corrected form of invalid numbers.

```
+ 1313
 123,456
 113.
 7b86.0
 .189
 19.E+11
  E 5
−11.11E + 6.0
+27.8 to 9E + 15
```

3.7. What do you understand by the term identifier? Give the Pascal language rules to design an identifier. Illustrate by examples. (An identifier can consist of how many characters on your system?).

3.8. What do you understand by Data Type? List all the data types available in Pascal. Give reasons for their existence.

3.9. Explain the differences between constants and variables. How are they defined? Give examples.

3.10. Consider the declarations

(a) **const**
```
    SELDOM = false ;
    TEN =  10 ;
    DOT = '.';
```
How does the Pascal compiler know that SELDOM is of boolean type, TEN is of integer type while DOT is of character type?

(b) **const**
```
    TEN = 10;
    X = TEN;
```
Is it valid to write like this?

3.11. Identify errors in the following declarations:

```
  (i) const
        MAXVALUE : 10000;
        PIE = 22/7;
 (ii) type
        X, Y = real;
        Y, Z = integer;
(iii) var
        P, Q = char;
        7 R : integer;
        T = boolean;
```

3.12. (a) Declare the following variables of the type indicated:

J, K, L → integer,
A, B, C → real,

LEG, LOG → boolean,
X, Y → character

and make the appropriate assignments (to any variable) with the constants:

6, 'HORSE', 1515, 9.8, 81, **false**,
'BABY', 1313, − 66.77, **true**

(b) Explain the difference between

CH: = P

and

CH: = 'P'

(c) Bring out the differences of declarring variable type with **type** and **var** declarations.

3.13. Explain the concept and importance of built-in functions. Prepare a list of all the arithmetic functions for real and integer data arguments.

3.14. When are the functions **trunc** and **round** generally used? Find the value of the following:

(i) **abs** (**trunc** (−36.9))
(ii) **round** (+ 14.6)
(iii) **trunc** (**sqrt** (**round** (3.4)))
(iv) **abs** (**round** (13.46)−**round** (18.67))

3.15. Explain the action of the following functions :

- **odd** (y)
- **eoln** (file-name)
- **eof** (file-name)
- **ord** (character)
- **chr** (ordinal-number)

3.16. What are the type of arguments for which the functions **pred** and **succ** are defined? Illustrate their use by examples.

3.17. State the result of the following :

ord ('P')
ord ('5')
chr (13)
pred (**ord** (9)))
succ (**true**)
succ (9)
chr (**trunc** (**sqrt** (66.0))
succ (**round** (5.7))
chr (**ord** ('a') + 3)

3.18. What do you understand by ordinal number of a character? Find the ordinal numbers of all the characters as available on your system.

3.19. Explain the meaning of collating sequence of characters in a character set? Is it same foı all systems or it can vary?

3.20. List all the built-in functions which can take character data as their arguments. Explain their meaning and use.

3.21. Explain the advantage of writing **const** and **var** declarations separately and independently.

3.22. Indicate the order in which type, constant and variable declarations are given in a Pascal program.

Chapter 4

Expressions

Expressions are formed using constants, variables and functions along with operators and other symbols. An expression always has a value and this value can be obtained by following the rules of expression evaluation. The value may be numeric, logical, string or a set constant. Expressions, which yield numeric value, are the arithmetic expressions; the expressions, which return logical values, are the boolean expressions, while the character expressions give character constants and the set expressions yield set constants as their values.

We shall study the construction and evaluation of arithmetic and logical expressions in this chapter and defer the discussion of other type of expressions to later chapters.

4.1 Arithmetic Expressions

A valid arithmetic constant, a variable, a function or a combination of these, formed using arithmetic operators, according to rules of Pascal language, defines an arithmetic expression. The following type of arithmetic expressions may be defined:

- integer mode expressions
- real mode expressions
- mixed mode expressions

Let us study each of these separately.

Integer mode expressions

Such expressions are formed using integer type constants, variables, functions and operators. The operators defined for integer data types are

Addition	+
subtraction	−
multiplication	*
integer division	**div**
modulus	**mod**

The operator **mod** operates as follows: let a and b be two integer quantities. Then a **mod** b gives the remainder after dividing a by b. Thus

7 **mod** 2 = 1
−7 **mod** 2 = −1 (quotient is −3)
7 **mod** (−2) = 1 (quotient is −3)
(−7)**mod** (−2) = −1 (quotient is + 3)

The operator **mod** is quite useful for applications involving circular counting, as for example, counting time or number of months in a year. In time, the value of 59 minutes is followed by 1 hour 0 minutes. Similarly, the value of 11 months is followed by 1 year and 0 months.

Let us declare

```
var
  J, K, P, NUMBER : integer;
```

Examples of integer expressions are

```
−6
 7
 K+P * NUMBER
 15 div 5+13 * J
 −250 + 20 * 12
 sqr (J) + abs (K)
```

The operator **div** must have integer operands, otherwise error will occur.

Real mode expressions

Real expressions are formed using real mode quantities: constants, variables, functions and operators. The operators defined for real data types are:

addition	+
substration	−
multiplication	*
division	/

Assume that X, Y, Z, PRICE are real variables:

```
var
  X, Y, Z, PRICE : real ;
```

Examples of real mode expressions are

```
13.13
7.1+9.8 * 3.0
X−Y / Z
PRICE / X+Y* sqrt (Z)
exp (X+Y)+ sqr (PRICE)
```

See that only real mode quantities have been used in these expressions.

Mixed mode expressions

Pascal permits mixing of integer and real quantities in an expression. Such an expression may be characterized as mixed mode expression. Examples are

```
6+7.0
J−X * P
ln (Y) + sqr (K)
PRICE / 9.2 + 6 div 4
```

Further examples of writing Pascal expressions for algebraic expressions, using the following declarations:

```
var
    H, M, N, K : integer ;
    C, P, Q, X : real ;
```

are

Algebraic expression	*Pascal expression* †
$\frac{2.4}{-3+4.2}$	2.4/(−3+4.2)
$\frac{mp}{x} - \sqrt{q}$	M * P/X − **sqrt** (Q)
$h \sin(p) + \log_e(x)$	H * **sin** (P) + **ln** (X)

Let us consider the algebraic expression

$$p^m + k^7 \div 7.0$$

Pascal does not define the exponentiation operator. So this expression cannot be translated into Pascal expression straightway as was done earlier. The exponent terms are translated into Pascal form using log function. If $a^b = y$, then b. (log a) = log y, or y = exp [b. (log a)]. Thus

$$p^m = \exp(m \log p)$$

and

$$k^7 = \exp(7 \log k)$$

Hence the expression

$$p^m + k^7 \div 7.0$$

is translated into Pascal as

exp (M * **ln** (P)) + exp (7 * **ln** (K))/7.0

We have seen that several operators may appear in an expression. Pascal demands that two operators must never appear side by side in an expression. This can be avoided by the use of parentheses. The following examples illustrate the correct and incorrect use of operators.

Invalid	*Valid*
b+ − a	b + (−a) or − a+b
j * − k	j * (−k) or −k*j
p/−q	p/ (−q)

† Here we have used the same variable name in Pascal expression as it appeared in algebraic expression. It is only for convenience. You can use any valid variable name in the Pascal expression.

4.2 Evaluation of Arithmetic Expressions

All the variables appearing in an expression must have been assigned values before the evaluation of an expression is performed. The computer scans the expression from left to right and finds its value. The value is obtained by combining the values of the items that constitute the expression. For instance, the value of the expression 3.1 + 4.5 is 7.6. It is obtained by adding the constant 4.5 to the constant 3.1. Assume a = 10.0, b = 15.0, c = 7.5. Consider the expression

$$a + b/c$$

The value of this expression is 10.0+15.0/7.5 = 10.0+2.0 = 12.0. First, b/c is computed and the result is added to the value of a.

Parentheses may be used in an expression. An example is

$$\underbrace{(a + b * c)}_{i} - \underbrace{(e/f - g)}_{ii}$$

When parentheses are used, the number of the opening parentheses must be equal to the number of closing parentheses.

In the above example, first the expression in paretheses (i) is evaluated and then the expression in parentheses (ii) is computed. The calculated result of (ii) is then subtracted from the result of (i). This is an example where one set of parentheses does not contain another set of parentheses. When such is the case, the evaluation of parentheses proceeds from left to right.

Parentheses may also be contained within one another. This is called Nesting. An example is

$$\underbrace{(x + \underbrace{(\underbrace{(a + b)}_{(i)} * c/d)}_{(ii)} - p * q)}_{(iii)}.$$

Parentheses (i) are nested within parentheses (ii), while parentheses (ii) are nested in parentheses (iii). First, the expression in parentheses (i) is evaluated; then the expression in (ii) is computed; and last of all, the contents of parentheses (iii) are calculated. Thus, when parentheses are nested, their evaluation is carried out from the innermost set, going outwards.

When a function appears in an expression, it is evaluated first and then the above procedure of expression evaluation is followed. Consider the expression:

a + b * **sqrt** (c)

Value of **sqrt** (c) is calculated first. It is multiplied by the value of variable b. The result of computations of b * **sqrt** (c) is added to the value of variable a.

It is obvious from the above examples of expressions that several operators may appear in an expression. When this is the case, how does one know as to which operator is to operate first, which one to operate next, and so on. Look at the expression

$$4 + 3 * 2$$

Its value is 10. This is because multiplication was performed first and then addition. If addition is carried out first and then multiplication, result would be 14 (?). What result is obtained depends on the order of operation of the operator. The priority of operator operation in an expression iş decided by hierarchy rules. These rules for arithmetic operators are given below:

mod	: first priority
* / **div**	: second priority
+ −	: third priority

Evaluation of integer mode expressions

We know that integer entities and operators occur in integer expressions. The result obtained is also an integer quantity. To illustrate the process of expression evaluation, we proceed as follows. Assume

```
    var
       K, P, M, N, J: integer ;
```

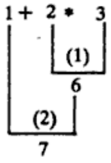
and

K = 1, P = 2, M = 3, N = 4 and J = 5

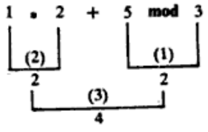
(symbol └─┘ indicates evaluation, (n) gives order)

Integer expression *Evaluation*

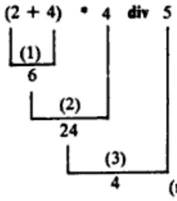
K + P * M



K * 2 + J **mod** M

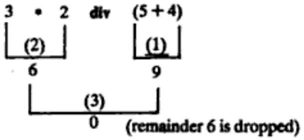


(P + N) * 4 **div** J



(remainder 4 is dropped)

M * P **div** (J+N)



N^J

$4^5 \rightarrow 4*4*4*4*4 \rightarrow 1024$
repeated multiplication

Generally, exponents of integer quantities are evaluated by repeated multiplication.

Evaluation of real mode expressions

These expressions involve real mode quantities and the final value of the expression is a real constant. Evaluation of a real expression follows the same procedure as for the integer mode expressions. However, in this case, fraction of a number is not dropped during the division process as was the case with integer mode division.

Expressions of the form

$$(-\text{base})^{\text{fractional quantity}}$$

are not permitted in Pascal. This is because negative quantity raised to a fraction is not defined for real quantities. Moroever, remember that *complex numbers and variables are not defined in Pascal.* Their manipulation is explained in Chapter 11.
Assume

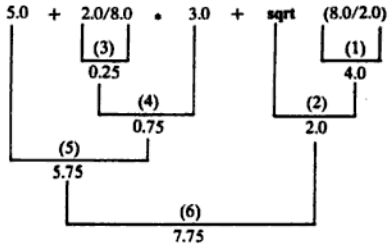
```
    var
      A, B, C, D : real;
```

and

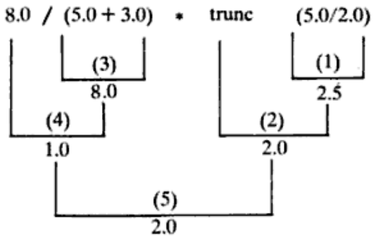
A=5.0, B=8.0, C=3.0, D=2.0

Real expression | *Evaluation*

A + D/B * C + **sqrt** (B/D)



B/(A+C) * **trunc** (A/D)



Evaluation of mixed mode expressions

In such expressions both integer and real entities occur. *Mixed mode expressions are the only exception where mixing of data types is permitted.* Evaluation of mixed mode expressions may be sometimes confusing and should be understood clearly.

Let us define the variables as

```
  var
     J, K, P : integer;
     A, B, C : real;
```

Assume J = 2, K = 3, P = 4 and A = 2.5, B = 3.5 and C = 4.5

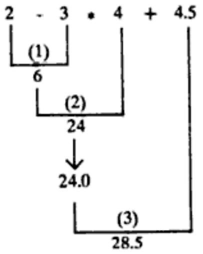
Consider the arithmetic expression

A + J

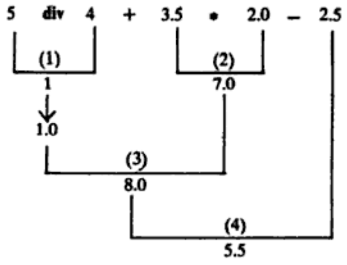
It is evaluated in the following way. Prior to taking the sum, the integer value 2 is automatically converted into the real value 2.0 and then the sum 2.5 + 2.0 = 4.5 is obtained. The value of J, that is 2, which is stored in the computer memory, remains in the integer mode. Conversion of the integer value to the real mode takes place only at the time of performing an operation. Further examples of mixed mode expression evaluation are: (symbol ↓ indicates conversion)

Expression *Evaluation*

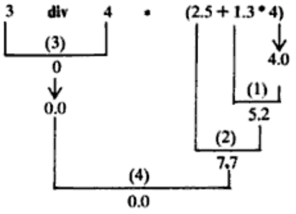
J * K * P + C



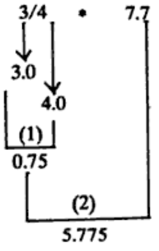
5 **div** P + B * 2.0 − A



K **div** P * (A + 1.3 * P)



K/P * (A + 1.3 * P)



What do you infer by comparing the evaluation of the last two expressions?

Remember, real mode and mixed mode expressions yield real values while integer mode expressions give integer values after their evaluation.

4.3 Boolean Expressions

You have learnt that there are two logical values: **true** and **false.** These values are always obtained from boolean (logical) expressions. The simplest form of a logical expression is a single logical constant, a single logical variable or a function. More complicated boolean expressions are constructed from the combination of these quantities using logical operators. Logical expressions can also be formed by joining arithmetic expressions with relational operators.

Logical Operators

Logical operators available in Pascal are: **not, and, or.** The definition and meaning of these operators are given below. (Assume that X and Y are boolean quantities).

Logical Operator	*Meaning*
not	If X is true, then **not** X has the value false. If X is false, then **not** X is true
and	If X and Y are both true, then X **and** Y has the value true. If either X or Y is false, then X **and** Y is false.
or	If either X or Y or both are true, then X **or** Y has the value true. If both X and Y are false, then X **or** Y is false.

These operations of logical operators are summarized in Table 4.2.

Table 4.2: Action of logical operators

X	false	false	true	true
Y	false	true	false	true
not X	true	true	false	false
X **and** Y	false	false	false	true
X **or** Y	false	true	true	true

Operators **and** and **or** always join two logical expressions, while the operator **not** has only one logical expression following it. Operators **and** and **not**, **or** and **not** may appear together in an expression

When only these logical operators appear in an expression, their hierarchy of operations is as:

Operator	*Priority*	*Explanation*
not	first	Logical expression after **not** is evaluated first.
and	second	Next logical expressions around **and** are evaluated.
or	third	Last of all, expressions around **or** are computed.

Boolean expressions using logical operators

Let us declare, B, P, Q, R, T as the boolean variables

```
var
  B, P, Q, R, T : boolean;
```

Examples of boolean expressions are:

> **false**
> P **and** Q
> Q **or** T
> **true**
> P
> (P **and** T) **or** (**not** Q)
> (**not** R) **and** (**not** B)

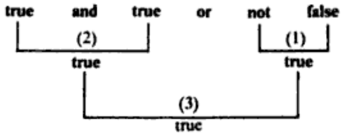
Parentheses may also be used in boolean expressions, just as in arithmetic expressions, to indicate operator priority in the evaluation process and for clarity.

We present below some examples to illustrate the evaluation of logical expressions. Assume that the logical variables P, Q, R, T have the following values:

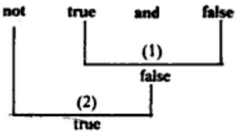
> P = **true**, Q = **false**, R = **false**, T = **true**

Boolean Expression *Evaluation*

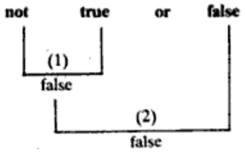
P **and** T **or not** R



not (T **and** R)



not T **or** Q



4.4 Relational Expressions

There are six relational operators available in Pascal. These operators can be used to compare the magnitude and ordering of quantities. Expressions formed by using them are called Relational Expressions. Such expressions return the logical values: true or false.

The relational operators, along with their meaning and hierarchy of operations, are given below:

Relational Operator	*Meaning*	*Priority*
=	equal to	highest
< > (≠)	not equal to	
<	less than	↓
< =	less than equal to	
>	greater than	
> =	greater than equal to	lowest

The form of the relational expression becomes

expression-1 $\boxed{*}$ expression-2

where $\boxed{*}$ is any of the above relational operators. Expressions on either side of relational operator may be either arithmetic expressions or character expressions and may be of any desired degree of complexity. However, it is necessary that the type of the two expressions must be identical. Relational boolean expressions are also referred to simply as Boolean expressions.

Examples of relational boolean expressions using arithmetic expressions are given below:

Let us declare variables

```
var
  Q, M, N : integer;
  A, B, C, D : real;
```

Examples of relational expressions are

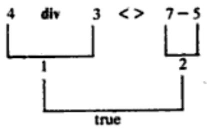
```
4 > 3
A + C < > B
Q * M − N div 6 > = M * N + 1 3
(A/2.0 + B/3.5) * D + 6 < = sqr (D) * 3.4E + 02
```

To illustrate the process of evaluation of relational expressions, let us assume that the values of variables are as:

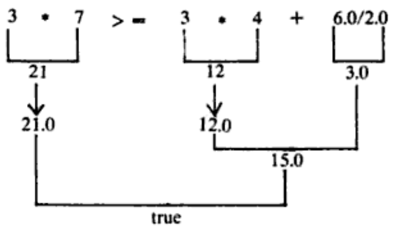
Q = 7, M = 4, N = 3;
A = 2.0, B = 4.0, C = 6.0, D = 7.0;

Relational expression — *Evaluation*

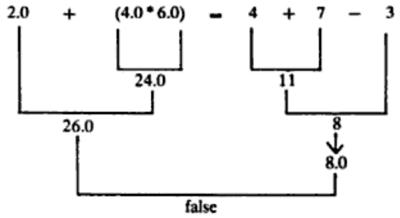
(i) M **div** N < > Q−5



(ii) N * Q >= 3 * M+C/A



(iii) A + (B * C) = M+Q−N



You will observe from example (ii) that integer and real expressions are being compared. When this is the case, the integer value is first converted to real and then comparison is made. Remember, the evaluation of relational expressions goes as follows: first, the expressions appearing on the two sides of the relational operator are evaluated according to the usual rules of calculating the arithmetic expressions and then the relational operator operates.

Examples of boolean expressions using both types of above operators and their evaluation have been brought out by the following illustrations.

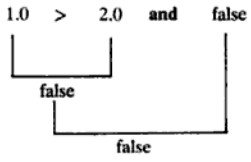
We declare variables as

```
var
  M, N, K : integer ;
  A, B, C : real ;
  P, Q, R, T : boolean ;
```

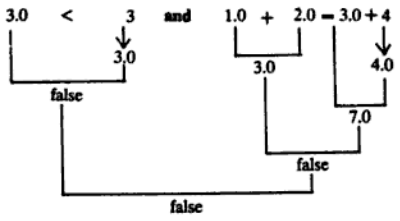
and suppose their values are as: M = 2, N = 3, K = 4; A = 1.0, B = 2.0, C = 3.0 and P = **true**, Q = **false**, R = **true**, T = **false.**

Logical Expression *Evaluation*

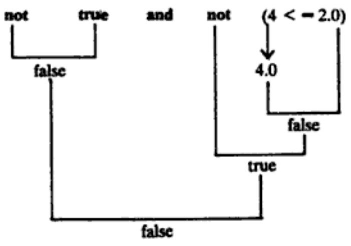
A > B **and** T



C <= N and A+B = C+K



not R **and not** (K <= B)



You have observed that a variety of operators can be used in expressions. The order in which operations are performed is summarized below:

not
*** / div mod and**
+ − or
=, < >, <, <=, >, >=
in (see Chapter 13)

The sequence of operations within an expression is from left to right following the above hierarchy rules.

We have described the construction and evaluation of expressions involving arithmetic and logical quantities. Expressions may also involve character, enumeration or set type of data. The presentation of these is deferred to later chapters.

Exercises 4

4.1. Tick the correct answers:

(a) An expression always has a single/multiple value(s).
(b) The **mod** operator can/cannot be used with real quantities.
(c) The value returned by a mixed mode expression is of the type real/integer.
(d) Exponentiation operator is available/not available in Pascal.
(e) Two arithmetic operators may/may not appear next to each other in an expression.
(f) Evaluation of arithmetic expressions starts from left/right.

(g) Nesting of parentheses in an expression is allowed/disallowed.
(h) The operators **and not** may/may not appear together.
(i) The relational operators join two logical/arithmetic expressions.
(j) ·The real and integer mode expressions can/cannot be compared.

4.2. Complete the following sentences

(i) Expressions can be formed with .... and .... using Pascal ....
(ii) Arithmetic expressions yield .... values.
(iii) Boolean expressions yield .... values.
(iv) The mode of arithmetic expressions may be ...., ..., and ....
(v) When parentheses are nested, their evaluation is carried out from .... to ....
(vi) Exponents of integer quantities are evaluated by ....
(vii) Mixing of data type is allowed only in .... expressions.
(viii) Boolean expressions may be designed using ....and .... operators.
(ix) The value of a function appearing in an expression is evaluated .... of all.
(x) The operator **div** is used for .... division.

4.3. Define an expression. Describe the various types of arithmetic and boolean expressions available in Pascal.

4.4. (a) Explain the action of **mod** and **div** operators.

(b) Evaluate

(−47) **div** 4
(−47) **mod** 4
(−47) **mod** (−4)
(−29) **mod** (−15)

4.5. Design equivalent Pascal expressions for the following algebraic expressions. (Take a, b, c, d, q, t, x as distinct symbols of real type).

$abc \div t + x$
$\sin(qt - \sqrt{ab}) \log(x + bc)$
$at - (ab \div c + x t^2)$
$\left(\frac{a+b}{c-d}\right) (q(x-t) \div (x-4))$
a^{bc}

4.6. Let i, j, k be integer variables with values 5, 4, 3 respectively. Find the values of the expressions :

i **div** j + 7
j **div** k − 8
i + 6 * j − (i + j + k)
j **mod** k
i * j **div** k **mod** 3
(7 **div** 2) **div** 2

4.7 Evaluate the following integer expressions

4 + (7−5) * 3
9 **div** 2 + 15 **mod** 6
3 * 5 + **succ** (16 **mod** 3) **div** 4
pred (18 **mod** 4) + **sqr** (**succ** (8))
5 + (9 * 10 **div** 3 + (6-2) * 7)

4.8. Assume that the variables X, Y, G, F, H are real and their values are: X = 2.0, Y = 3.0, F = 4.0, G = 5.0 and H = 1.0. Evaluate the following expressions:

```
G + trunc (F/X) + 8.8
17 mod (−5) + X * F/ sqrt (F)
F − F * Y (1.0/G − trunc (F + 6.7) )
X + Y round (F/2.5) − G/F
sqrt (F + (X+Y)/F − 1.0)
7 div 3 * F + Y/X
```

4.9. List the various boolean and relational operators available in Pascal. Explain their meaning.

4.10. How are relational expressions formed? Illustrate by examples.

4.11. Assume the following declarations

```
var
  P, Q, T : real;
  J, K, L : boolean;
```

Find mistakes in the following expressions, if any, and write their correct forms appropriately.

```
P and J
(P+Q) > T/ 2.13
J and K not L
not (P+Q < > K)
P div Q >= − T / −F
```

4.12. Take the following declarations and values:

```
var
  A, B, C : boolean ;
  X, Y, Z : real ;
  M, N : integer ;
```

A = true, B = true, C = false; X = 2.0, Y = 3.0 and Z = 4.0; M = 6, N = 7

Evaluate the expressions give below:

```
A and C or B
not B or C
( X < Y ) and (Y + Z < > X − 6)
not ( Z − 2 * X/Y < 2.0 * Y * Z)
not A and not C and not B
not odd (N) and E
( X > 3.0) or ( Y > 4.0) and ( Z > X )
```

4.13. Evaluate the following real and mixed mode expressions:

```
2.4 * 5.0 + abs ( 16.2 − 4.0 * 6.1)
1.0 + 2.0 * 3.0 / 5.0 + trunc (−18.0/4.0)
2 + 3 div 2 + pred (10 mod 4)/5.0
3 + sqr (4.0) * 3
round (sqrt ( 28.0/5 )
ord ('D') * pred (7)/9.0
```

4.14. What do you understand by hierarchy of Pascal Operators? Compare it as given in the text and the Pascal system implemented on your computer system. Bring out the importance of operator hierarchy.

Chapter 5

Simple Statements and Programs

Variables occur extensively in expressions. They must be assigned values before the expressions can be evaluated. When a variable has been given a value, it is said to be defined or initialized. Computers give an error message if any variable in a program is undefined during the execution phase. Thus, all variables occuring in a program must be carefully defined. A variable is defined either by an assignement statement or by an input statement. The input statement reads value for the variable from an input device. The assignment statement assigns a value to the variable after evaluating an expression.

We shall examine the assignment and input/output statements in this chapter. Labels and comments will also be introduced here. This will enable us to develop simple programs at an early stage. All Pascal programs have a specific structure and is explained below:

5.1 Structure of a Pascal Program

Every Pascal program must have three parts:

- program heading
- declaration part
- execution part

These must appear in the order indicated.

(a) *Program Heading*

The program heading consists of a single statement whose format is

program *name* ($f_1, f_2, \ldots\ldots f_n$)

where

program	reserved word,
name	user-defined name by which the program will be known
$f_1, f_2, \ldots\ldots f_n$	names of external data files used by the program to communicate with the outside environment; their specification is optional.

The reserved words **input** and **output** are standard file names. They are used to specify that the program needs input (data) and will yield output (results) on the output device. Thus, an example of statement **program** is

program INTEREST **(input, output)**

Here INTEREST is the name of the program which needs input data and will give output.

The **program** *name* statement may be used without indicating **input, output** specification. This would imply that the program does not need input data and will not yield output on the external device. If no input data is required, but output results are expected we can specify

program SUM **(output)**

(b) *Declaration Part*

All the constants, variables, identifiers, labels, subprograms, etc., appearing in a program, must be declared and specified prior to their use in the execution part of the program. This is done in the declaration part of the program which must immediately follow the statement **program.** The declarations which appear in the declaration part are: **label, const** (constant), **type, var** (variable), **function** and **procedure.** Thus, the declaration part appears are

label declarations;	(i)
const declarations;	(ii)
type declarations;	(iii)
var declarations;	(iv)
function declarations;	(v)
procedure declarations;	(vi)

There are six declaratons and they must appear in the order indicated except that function and procedure may be interchanged. You have learnt about **var** declaration and will be learning about others as we go along. Semicolon follows all declarations to separate them from each other.

In the declaration part of the program, we specify information about the data types, attributes of data objects, value of a data object (if it is a constant), name of the data object, labels and so on. The declaration part serves to communicate to the Pascal compiler information about the various entities needed during execution in the program body and the possible operations associated with them. This enables the compiler to:

- perform type checking,
- determine the optimum representation for the data objects,
- organize more efficient storage management,
- determine the meaning of the operator symbols when the same symbol is used with different data types, as for example "+" is used for arithmetic addition and set union.

(c) *Execution Part*

The execution part of the program consists of all the executable statements. All computations/processing are performed in this part. Data are supplied here, used in computations, and results obtained. The start of this section is indicated by the reserved word **begin** and end by the reserved word **end** which is followed by the period (.) Denoting any statement by the letter *S*, the structure of the execution part appears as

```
begin
   $S_1$ ;
   $S_2$ ;
   :
   :
   :
   $S_n$
   end.
```

Each statement is followed by a semicolon. In the case of the last statement, S_n, it is optional. Remember: *semicolon is used as a separator and not as a terminator of a statement / declaration. Moreover, there is no semicolon after the* **begin** *or before the* **end** *words.*

Let us indicate declarations appearing in a program by *D*'s.

The complete structure of a Pascal program may be represented as

```
program name (. . . . , . . . , . . .) ;
   $D_1$ ;
   $D_2$ ;
   :
   :

   $D_n$ ;
begin
   $S_1$;
   $S_2$ ;
   :
   :

   $S_n$
end.
```

Declarations and statements may be placed, either one or more than one, on a single line.

```
program name (. . . ., . . . .) ;
   $D_1$; $D_2$ ; . . .;

   begin $S_1$ ; $S_2$ ; $S_3$ ;
        $S_4$. . . ., $S_n$ end.
```

The reserved word/identifier or the number must not be divided between lines. If there is not enough space on a line for the complete identifier or number, this may be started on the next line. *All blanks in a statement/declaration are ignored except within the names and numbers.*

The declarations, *D*, and statements, *S*, may be indented to make the program better readable and clear. This will become clear as we go along.

The above structure of Pascal program offers many advantages, such as:

- structure of all Pascal programs is uniform,
- any changes in the execution part do not affect the declaration part,
- program efficiency enhances as the compiler can decide the storage mangement and execution strategies at the compile time itself based on the information specified in the declaration part,
- program modifiability improves,
- communication with the outside environment is only via the program heading statement specification,
- program portability (from one computer system to another) enhances as most of the changes are generally required to be made in the declaration part and that is easy to do,
- program development process is also simplified as the programmer can plan the declaration and execution parts separately and then join them appropriately.

5.2 Comments in a Program

Explanatory notes may be introduced into a program. They may identify the various parts of a large program, help the reader in understanding the flow of the program and what is being calculated at various stages.

Comments in a program are specified with the use of braces { }. Its form is

{ *text* }

An example is

{ This program computes the roots of a quadratic equation. }

The *text* does not have to be enclosed within quotes. The matter enclosed between { } is ignored by the Pascal compiler. It is meant for readers as a documentation aid.

In some Pascal implementations, symbols (* are used for left and *) for right braces. However, we shall always use braces to denote comments in a program.

5.3 The Assignment Statement

The form of an assignment statement is:

v := *an expression*

where *v* represents an unsigned simple or subscripted variable. The symbol := is called the Assignment operator.

The *expression* appearing on the right hand of sign := may be

- an arithmetic expression
- a boolean expression
- a character expression
- a set expression

When the expression is arithmetic, then we have an arithmetic assignment statement. When the expression is boolean, then the statement is boolean assignment statement. Similarly, for the case of character and set expressions.

The operator := (appearing between the variable of left side and the expression on the right side) assigns the value, obtained by evaluating the expression, to the variable. Any previous value of the variable is lost. The type of variable must be same as the type of expression, except for one exception which you will see.

The assignment statement is separated from other statements by a semicolon.

Arithmetic Assignment Statement

This is specified as

v := an *arithmetic expression*

v represents an arithmetic variable identifier, may be simple or subscripted.

Declare

```
var
  X, A, B, C : real ;
  J : integer ;
```

Examples of arithmetic assignment statement are:

- X := 3.4 ;

The number 3.4 is made the value of X.

- A:= 2.0+3.0 * 4.0 ;

The expression 2.0+30 * 4.0 is evaluated. Its value is 14.0. This value is assigned to variable A.

- J := 3 **div** 2 * 4 ;

The value of expression 3 **div** 2 * 4 = 4 is assigned to J.
Now consider

```
A := 3.6;                                    (i)
B := 2.0 * A - 1.2;                          (ii)
C := 3.4;                                    (iii)
A := A+B+C;                                  (iv)
```

Here, statements (i) — (iv) are assignment statements. In line (i), A is assigned the value 3.6. Thus, A is initialized to 3.6. In line (ii), value of the expression 2*A − 1.2 = 2.0 × 3.6 − 1.2 = 7.2−1.2 = 6.0. This becomes the value of B. Next, C is set to the value 3.4 in line (iii). In line (iv), the expression A+B+C = 3.6+6.0+3.4 has the value of = 13.0. This value is given to variable A. The previous value of A (i.e. 3.6) has been lost. If now onwards, variable A appears in an expression, its value will be 13.0.

The reader will note that two actions take place when an assignment statement is executed. First, the value of the expression on the right is obtained and then this value is assigned to the variable on the left. It can happen that the mode of value of the expression and the mode of the variable identifer are different. The variable may be of integer type and expression of real type and vice versa. This is the only type exception allowed in Pascal about which we talked earlier. However, the mode of storage of the value in the computer memory is determined by the type of variable, irrespective of in what mode the value of the arithmetic expression is obtained.

A word of caution here. There is no difficulty when the variable is real while the value of the expression is integer. The integer value will be transformed into the real mode. But when the value is real and the variable is integer, then the value may be assigned either after truncation or rounding. What happens exactly may be ascertained by reference to the installation Pascal manual.

The value of a variable is changed only when it appears on the left-hand of an assignment statement. Its appearance on the right-hand side does not alter its value, but the value is used in the expression. For instance, consider the statement

```
J:=J+1
```

Here the variable J occurs on the left and in the expression on the right hand side. As there are two separate actions performed by the assignment statement, so this statement is executed as : first the expression J + 1 is computed using the current value of J. The new value, obtained thus, is assigned to J. The previous value of J is no more available. We can represent all this as:

new value of J = old value of J + 1

Boolean Assignment Statement

The form of this statement is

v := a *boolean expression*

where *v* represents a logical variable, simple or subscripted.

Let us declare X, Y, Z, W as boolean variables and A, B, C as real:

```
var
   X, Y, Z, W : boolean ;
   A, B, C : real
```

Examples of boolean assignment statements are:

```
X: = true ;
Y: = false ;
Z: = not ( X and Y ) ;
W: = (A+B*C > A*5.0−B/C) or (A*B*C <= A*B−C)
```

A program is run to perform computations and process data on the computer. During its execution the program may need data and generate results of the calculations. The data required by the program can be supplied by input statements while the results of processing/computations can be obtained with the help of output statements. Thus, data transmission between the computer and the peripheral devices is carried out with input/output statements. The input operations are performed with two statements-**read** and **readln**, while output operations are carried out with **write** and **writeln** statements. In order to introduce the student to the process of writing programs at an early stage, we shall discuss the output statements first and then go over to the input statements. Moreover, we shall always assume that a visual display unit (VDU) with a keyboard, is used for input of data while for output, both printer and VDU are available.

5.4 Data Output Statements

As explained earlier, the output statements provide a means of obtaining information from the computer on an output device. This information may be in the printed form, displayed on the screen, and so on. We can also have the output information in a properly formatted form. All this is done with the output statements **write** and **writeln.** Let us first study **write** statement.

The **write** *statement*

A form of this statement is

write (output, *list*)

where

write	is the key word
output	is the keyword which indicates that the values to be output form an output file; its use is optional
list	may be constants, variables, expressions, messages (text), separated by comma.

The keyword **output** and the *list* must be enclosed within parentheses. Specification of **output** and *list* is optional and may be omitted when no output is desired.

When the **write** statement is executed, the values of items appearing in the *list* are written in the same order, as they are specified in the *list*, on the output device.

Consider the statement

```
write (output, 3,−4.4, 13.13)
```

According to this statement, the constants 3, −4.4, 13.13 are written on the output device in the order

```
3ƀ−4.4 ƀ 13.13
```

on the same line. Each value is separated from the other by a single space (ƀ).

The statement

```
write (A, SUM, 5.15)
```

will write the values of variables A, SUM and constant 5.15 on one line.

With the statement

```
write ('TIME  FOR  TEA', 7, 'A.M.')
```

the output will appear as:

```
TIME FOR TEA 7 A.M.
```

Expressions may be used in the *list* of **write** statement. Values of expressions are obtained first and then written.

The statement

```
write (3+4*5, sqrt (8.0*2.0) + trunc (6.7) )
```

will give the output as

```
23   10.0
```

See that 23 is the value of expression 3+4*5 and 10.0 is the value of (**sqrt** (8.0*2.0)+**trunc** (6.7)).

The **write** statement operatres as follows. The data to be output are stored in a memory buffer. When the **write** statement is executed, its contents are transferred to the output device. The output appears on a single line. If the values exceed a line, the output goes to the next line. The **write** statement does not automatically cause the return carriage so that output from the next **write** statement can begin from a new line. For instance, consider the appearance of the two **write** statements in a program as:

```
write  ('A =', 2.0, 'B =', 3.0, 'C ='. 7.56 'SUM =', 2+3+7.56) ;
...............
...............
...............
...............
...............
```

```
write ('CALCULATION OF INCOME TAX');
...............
...............
...............
...............
...............
```

the output will appear as

```
A = 2.0 B = 3.0 C = 7.56 SUM = 12.56 CALCULATION OF
INCOME TAX
```

This is a limitation of the **write** statement which has been overcome by a slight variant of this, that is, **writeln** statement.

The **writeln** (write line) statement is same as **write** statement with the additional feature that carriage return is generated automatically when a line has been written on the output device from the memory buffer. The new data automatically goes to the next line.

The statement

writeln (output, ***list*****)**

is equivalent to

```
write (list) ;
writeln
```

Examples of **writeln** statements are

(i) **writeln** ('A = ', 10 * 10.4, 'J =' 13+12) ;
(ii) **writeln** ('SUM OF FIRST 10 PRIME NUMBERS IS 101')

Output of line (i) appears as

```
A = 104 ƀ J=25
```

and that of line (ii) as

```
SUM OF FIRST 10 PRIME NUMBERS IS 101
```

When the **writeln** statement is specified without arguments, then a single line is skipped by its use. Suppose, you wish to skip three lines, then use **writeln** statement three times as:

```
writeln ;
writeln ;
writeln
```

Thus, the **writeln** statement also helps to control the vertical arrangement of lines in the output.

Let us write a program and illustrate the use of **write** statement.

Example 5.1

The distance, d, travelled by an object moving with a velocity, v, having acceleration g, after time, t, is given by

$$d = vt + \frac{1}{2} gt^2$$

Develop a program which computes the distance travelled when v = 25.0 metres/sec, t = 50.0 seconds and g = 9.8 metres/sec^2

The process of developing the program may be as follows:

(i) assign a name to the program.
(ii) declare the variables: d, v, t and g as real (because the given values are in real mode)

    var D, V, T, G : **real**

(iii) assign values to the variables v, t, g as

    V = 25.0; T = 50.0; G = 9.8

(iv) prepare Pascal expression for the relation $vt + \frac{1}{2} g t^2$:

    V * T + 0.5 * G * **sqr** (T)

or as

    V * T + 0.5 * G * T * T

(v) write the assignment statement as:

    D:= V * T + 0.5 * G + **sqr** (T)

(vi) write the values of V, T, G and D using the output statement **writeln** including appropriate titles.

Thus, the complete program appears as:

Program 5.1

```
program DISTANCE (output) ;
   var
      D, V, T, G : real ;
   begin
      { Assign values to variables }
      V := 25.0 ;
      T := 50.0 ;
      G :=  9.8 ;
      writeln ('VELOCITY = ', V);
      writeln ('TIME =', T) ;
      writeln ('ACCELERATION = ', G) ;
      D := V * T + 0.5 * G * sqr (T) ;
      writeln ('DISTANCE =', D)
   end.
```

Sample output

```
VELOCITY = ƀ 2.500000 E + 01
TIME = ƀ 5.000000 E + 01
ACCELERATION = ƀ 9.800000 E + 00
DISTANCE = ƀ  1.350000 E + 04
```

The above output has been shown upto six places so the right of the decimal. This may vary from system to system.

5.5 Formatting of Output Data

Look at the integer number 23456. It has five digits. We say its width is 5. Similarly, width of the number −1698134 is 8 (including the minus sign). Width of a number is counted along with its sign. Plus sign may or may not be counted. The width of a character constant will be the number of characters in the constant, including the blank space, if any.

Sometimes, it may be desired that the integer number be written upto a certain width. This can be done with a **write** statement by indicating the width, *w*, of the constant or variable, within the parentheses, as

write (*expression*-1 : w_1, *expression*-2 : w_2, . . .)

or

writeln (*expression*-1 : w_1, *expression*-2 : w_2, . . .)

w is called the field width. It must be separated from the expression by a colon (:). *w* may be an integer constant, variable or an expression. Now w_1 controls the field width of value of *expression*-1, w_2 controls the width of value of *expression*-2, and so on. The width of each number is indicated by a separate width factor.

Consider the example

write (180 * 15 : 5)

This will output the number as

ƀ2700 $w = 5$

that is, a blank is inserted at the leftmost position to make the field width 5.

If we had specified

write (180 * 15 : 3)

the number written will be

700

that is, the leftmost digit 2 is truncated.

Let us consider the example

write (SUM: 6, NUMBER: 8)

This would write the value of variable SUM consisting of 6 digits and that of variable NUMBER having 8 digits (taking into account sign as well). The variables SUM and NUMBER have been assumed to be of integer type.

In the case of real numbers, there is a decimal part as well. For instance, in the number 13.267, .267 is the decimal part. For this number, width is 6 (including the decimal) while the number of digits to the right of decimal point is 3. Let us indicate the width of a real number by *w* (including sign and decimal) and the number of digits to the right of the decimal by *d*. When the numbers are written as per the given specifications, the output is said to be Formatted. We can have the real numbers written with a desired width and number of decimal places, with a **write** statement. The format of **write/ln** statement is, then,

write (*expression*-1 : w_1 : d_1, *expression*-2 : w_2 : d_2, ...)

w and *d* must be integer expressions and $w > d$. Look at the statement

write (4.0/3.0 : 6 : 3)

The output obtained will be

ƀ1.333 ($w = 6, d = 3$)

Similarly, the output of the statement

write (5.0 * 10.5/4.00: 4 * 2 : 4) {$w = 4 * 2 = 8, d = 4$} ← *w* is an expression

will be

ƀ13.1250

Remember: *w* and *d* may be integer constants or expressions.
The arithmetic results are right justified, that is, truncation or addition of blanks/zeroes takes place on the leftmost side.

When it is desired that the real number be printed in the exponent form, then the specification : *d* is not used. In that case, the **write/ln** statement appears as

write (*expression*-1: w_1, *expression*-2: w_2, ...)

The output value is obtained in exponent form in such a way that there are a total of *w* characters, including the sign of the number and the exponent. (In some implementations, the positive sign may be suppressed and replaced by a blank). As an illustration, look at the statement

writeln (20.0/6.0: 12)

The output appears as

3.333333E +00 {$w = 12$}

Another illustrative example is

write (−**sqrt** (13.0* **trunc** (5.3)) : 15)

Its output appears as

```
ƀƀƀ4.242640E+00          { w=15 }
```

The accuracy upto which a real number, in exponent form, is written depends on the computer system. Here, we have assumed that 6 places to the right of the decimal point are retained.

The formatted form of **write/ln** statements help to control the spacing between the output values. This is brought out further by the following examples.

The output of the statement

writeln (1000: 4, 99 : 2)

will be

```
1000ƀ99
```

one blank between the value 1000 and 99 is automatically supplied by the Pascal compiler. Suppose, you want that there should be 6 blanks. This can be done as

writeln (1000 : 4, 99: 7)

Here, the output will be

```
 1000ƀƀƀƀƀƀ99
w= 4↑      w= 7
     this blank is inserted by the default option
```

This way any horizontal spacing can be generated. Another method is by the use of blanks in character strings. For example, the output of the statement

write ('ALPHA =', 5*150: 5, 'ƀƀƀ ƀ', BETA =', 100)

appears as

```
ALPHA=ƀƀƀ750ƀƀƀƀBETA = 100
```

Example 5.2

Given a data value 7.62 feet. Convert this into inches and centimeters and output your result in a formatted form. One value should be written on one line.

Program 5.2

```
program CONVERSION (output);
{ declare the variables }

   var

      FEET, INCHES, CM : real ;

   begin
```

```
        FEET := 7.62;
        INCHES := FEET * 12.0;
        CM := INCHES * 2.54;
        writeln ('FEET = ', FEET: 5:2);
        writeln ('INCHES = ', INCHES: 6:2);
        writeln ('CMS =', CM: 7:2)
      end.
```

Sample output

```
FEET =ƀ 7.62
INCHES = ƀ91.44
CMS = ƀ 234.25
```

This program is self-explanatory and the reader should run it on his system.

5.6 Data Input Statements

We have seen how variables can be initialized by assignment statements. Another way of doing so is by the input statements. With the latter, data values are supplied via an input device. Statements **read** and **readln** are used for this purpose.

The **read** *statement*

A format of the **read** statement is

read (input, ***list*****)**

where

read	is the keyword
input	a keyword which indicates that incoming data form an input file; its use is optional
list	indicates variables, separated from each other by comma.

An example of **read** statement is:

read (A, B, C) (i)

This statement indicates that variables A, B, C will obtain their values from the input device. Similarly, the statement

read (input, ALPHA, BETA, GAMA) (ii)

implies that values of variables ALPHA, BETA, GAMA are to be supplied via an input device. Suppose these variables are to be initialized to the values 67, 78, 131 (assuming them to be integer variables). We can supply the values as :

```
67ƀ 78ƀ 131
```

or as

```
 67 |←  (The symbol  |←  indicates carriage return)
 78 |←
131 |←
```

When the **read** statement (ii) is executed, variable ALPHA gets the value 67, variable BETA is assigned the value 78 while variable GAMA is set to the value 131. The data values are always read in the order in which they are specified on the terminal (of input device). Values for the different variables are normally separated by blanks; alternately, one value may be written on one line. The pressing of carriage Return Key (|←) transfers data from the terminal to the computer memory.

In Pascal, the incoming data forms an input file. This file is 'divided' into lines. Data are 'written' on each line when the carriage return key is pressed. The consecutive data appearing on the lines become the value of consecutive variables specified in the list of the **read** statement.

Assume two **read** statements are specified as:

read (X, Y) ; (iii)

read (P, Q, R) (iv)

and we specify the data on the terminal as

```
3.4 ƀ 4.5 ƀ 5.6
6.7 b 7.8 ƀ 8.9
```

then the variables will get the values as X = 3.4, Y = 4.5, P = 5.6, Q = 6.7, R = 7.8. Here you will see that data item 5.6 appears on the first line but is still being used by a variable P of the second **read** statement. The two **read** statements (iii) and (iv) are equivalent to a single statement as

read (X, Y, P, Q, R)

Thus, while reading data with **read** staement, it is immaterial on which line data appears. However, the number of data values supplied must not be less than the number of variables in the *list* of the **read** statement

Another form of **read** statement is

readln (input, ***list*****)**

This statement is essentially the same as the **read** statement with the difference that any value after the data value, which is assigned to the last variable in the *list* of the **readln** statement, is ignored. For example consider the statements

readln (XX, YY, ZZ) ; (v)

readln (PP) (vi)

and the data are typed as

−13.66   151.67   56.123   94.1248

The variables XX, YY, ZZ will be assigned the values: −13.66, 151.67, 56.123 respectively when the **readln** statement (v) is executed. After this, the data line is skipped. When the statement (vi) is executed, no more data are available as the data of previous line becomes inaccessible.

These concepts are further illustrated by the following example. Suppose data values have been written as

```
3 4 5 6 7
8 9
```

When the statements

```
read (A, B, C) ;
read (E, F, G)
```

are executed, then the values assigned to the variables are A = 3, B = 4, C = 5, E = 6, F = 7, G = 8. However, if the statements executed are

```
readln (A, B, C) ;
readln (A, B, C) ;
```

then assignments are A = 3, B = 4, C = 5, E = 8, F = 9 and there won't be any value available to be assigned to variable G.

The **read/write** statements may be used to input/output numeric, boolean or character data. These are illustrated by the following examples:

```
var
  A, B : real ;
  U, W : boolean ;
  S, R : char
```

Suppose we wish to initialize A, B to 1.121 and −13.08 ; V, W to true and false ; and S, R to THE and BOOK respectively.

A possible form of **read** statement could be

```
readln (A, B, V, W, S, R)
```

and the values will be presented on the input device as

1.121 ƀ −13.08 ƀ true ƀ false ƀ THE ƀ BOOK

With the **read** statement, the type of data supplied must match the type of variable, otherwise error will occur

The output may be specified (say) as:

```
writeln (S, V, A : 5 : 3) ;
writeln (W, R, B : 6:2) ;
```

The printout of these two statements appears as

```
THEƀtrueƀ1.121
falseƀBOOKƀ−13.08
```

Example 5.3

Read any positive integer number $\leqslant$ 15. Determine whether it is odd or even. If the number is odd, print the result as TRUE, but if the number is even, output the result as FALSE. Obtain the binary representation of the number.

We can find whether the number is odd or even by using the **odd** function. The binary representation is obtained by the method of successive division by 2. As for example, suppose the number read is 13. Its binary representation will be given by R_4 R_3 R_2 R_1 where R's are:

$13 \div 2 = 2*6+1$ remainder (R_1)
$6 \div 2 = 2*3+0$ remainder (R_2)
$3 \div 2 = 2*1+1$ remainder (R_3)
$1 \div 2 = 2*0+1$ remainder (R_4)
Thus $(13)_{10} = (1101)_2$

The program appears as given below:

Program 5.3

```
program ODDEVEN (input, output) ;
  { Program to find whether the given
    number is odd or even and its
    binary representation }
var
  W: boolean;
  N, Q1, Q2, Q3, Q4 : integer ;
    R1, R2, R3, R4 : integer ;
begin
  writeln ('Enter the number <= 15') ;
  readln (N);
  W:= odd (N) ;
  writeln (N , 'is odd:', W) ;
  { Conversion to binary }
  Q1:= N div 2;
  R1:= N mod 2;
  Q2:= Q1 div 2;
  R2:= Q1 mod 2;
  Q3:= Q2 div 2;
  R3:= Q2 mod 2;
  Q4:=Q3 div 2;
  R4:= Q3 mod 2;
```

```
    writeln ('Decimal number =', N) ;
    writeln ('Binary representation =', R4, R3, R2, R1)
  end.
```

Sample input/output

```
Enter the number <=15
13 |←
13    is odd:      true
Decimal number = 13
Binary representation = 1 1 0 1
```

We have written the *Program* 5.3 in a simple way. It has some drawbacks, though output obtained is correct. In order to remove them we need to learn more Pascal statements. We shall return to this program again and again to remove its faults and illustrate the process of program refinement using Pascal language features. You try to locate the possible drawbacks here.

5.7 The label declaration

Any statement may be labelled. The label must be an unsigned integer number whose value may lie between 1 to 9999 (that is, a maximum of 4 digits). Labels must be declared in a declaration part as:

label l_1, l_2......

where l_1, l_2, ...are the integer constants of length 1 to 4 digits. Only these can be used as labels in the program. Labels are separated from a statement by a colon (:) as

l : *statement*

Examples are

```
.......
.......
label 13, 37 ;
.......
.......
.......
.......
13: writeln (X, Y);
37: AREA: = sqr (LENGTH) ;
```

Here 13 is the label of statement **writeln** (X, Y) and 37 is the label of the assignment statement AREA = **sqr** (LENGTH).

It is not necessary to maintain any kind of numerical order for the labels, but labels must be unique. For instance, it is wrong to write.

```
23 : read (RADIUS) ;
23 : write (RADIUS) ;
```

Statement label serves as an identifier and helps to transfer control in a program to the desired point. A label may or may not be referred to in a program. Normally, labels are used with a **goto** statement (chapter 7) , however, the use of this statement is always discouraged (Section . . .) The declaration **label** is optional, but if given, it must immediately follow the program heading declaration.

5.8 The Compound Statement

A set of statements enclosed within the reserved words **begin** and **end** is referred to as a Compound statement. Its structure appears as

```
begin
  S1; S2 ; . . . . ; Sn
end;
```

where S_1, S_2,. . .S_m are the Pascal statements.
An example is

```
begin
  readln (P, Q, R) ;
  SUM : = P+Q+R ;
  writeln (P, Q, R, SUM)
end.
```

Each of the statements S_1, S_2, S_3, ..., may again be compound statements. An illustration is

```
┌ begin
│   S1 ;
│   S2 ;
│   :
│
│   begin                          ┐
│     S11 ; S12; S13 ; . . . . .   │ ← compound statement
│   end                            ┘
│   S3 ;
│ :
└ end ;
```

The symbols **begin** and **end** are always specified in pairs, like the left and right brackets, as shown above. A compound statement may be labelled like any other statement. All statements within a compound statement are executable and no declarations are specified. It is supposed that all declarations have already been given in the declaration part of the program.

Example 5.4

You know that the roots of the quadratic equation

$$ax^2 + bx + c = 0 \qquad (1)$$

are given by

$$x_1 = (-b + \sqrt{b^2 - 4ac})/2a \qquad (2)$$
$$x_2 = (-b - \sqrt{b^2 - 4\,ac})/2a \qquad (3)$$

Develop a program which evaluates the roots x_1 and x_2 for $a = 2.0$, $b = 5.0$, $c = 1.0$.

The computation of roots can be performed easily using equations (2) and (3) after assigning values to the parameters a, b, c. For illustration, we group the statements that calculate the roots in a compound statement. The program may be as follows.

Program 5.4

```
program QUADROOT (input, output);
  { Program to calculate the roots of a quadratic equation }
  label 13, 14, 15. 16, 17 ;

  const
    TWO = 2.0;
  var
    A, B, C, D, DISC, ROOT1, ROOT2 : real;
  begin
    writeln ('Give values of A, B, C so that the discriminant is non negative')
    readln (A, B, C);
    { Roots are calculated below }

  13: begin
    14: DISC: =(B*B -4.0 * A *C) ;
    15:D: = A * TWO ;
    16:ROOT1 : = sqrt (- B + DISC) / D;
    17:ROOT2 : = sqrt (- B - DISC) / D
     end;
    writeln ;
    writeln ('A =', A:4:1, 'B =', B:4:1, 'C=', C:4:1);
    writeln;
    writeln ('ROOT1 = ', ROOT1 : 6:3) ;
    writeln ('ROOT2 = ', ROOT2 : 6:3)
end.
```

Sample input

Give values of A, B, C so that the discriminant is non negative

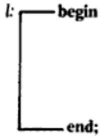
```
2.0   5.0   1.0
```

Output

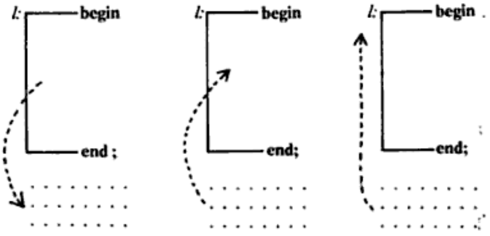
```
A = 2.0   B = 5.0   C = 1.0
ROOT1 = −0.219
ROOT2 = −2.280
```

The reader should appreciate the presence of the compound statement labelled 13 (for convenience) in Program 5.4. Say, for example, if any difficulty occurs in the program while computing the roots, we may refer to the compound statement, examine its statements first and then go to other parts of the program.

Control of program execution can be transferred out of the compound statement by the control statements but not into it from outside. Let us denote a compound statement as



where *l* is its label. Then the valid and invalid transfers are shown below:



Valid transfer   Invalid transfer   Valid transfer

These transfers are further illustrated in Chapter 7.

Compound statements are useful to group statements which are to perform some set of computations in a program. Use of compound statements, thus, imparts a sort of structure to the program body which is easy to understand, thereby, increasing the program transparency.

The compound statement, though a single statement, refers to a group of statements. Due to this, use of compound statements with transfer (Chapter 7) and iterative constructs (Chapter 8), greatly increases their utility and versatility and this further facilitates the structured design of computer programs (Chapter 15).

There is one big disadvantage also with compound statements. Statement/(s) within the compound statement cannot be accessed or referred to from outside. Refer to Program 5.4. Say, for instance, you want to reach statement labelled 15 by a transfer statement, this is not allowed. To reach any of the statements 14, 15, 16, 17 entry will have to be made at label 13. This involves execution of statements which we may not want. Thus, compound statements may introduce redunancy in a program but this may be avoided by duplicating that particular statement outside the compound statement, if the logic of the program permits.

In this chapter, we discussed the use of standard data types to develop programs. Before we take up the study of the control and iterative statements we would like to discuss next the availability of user-defined scalar data type in Pascal and illustrate their use in later chapters.

Exercises 5

5.1. *Answer as True or False:*

(a) A variable must be initialized before it is used in any expression during execution phase.(T/F)
(b) A variable cannot be initialized by an input statement.(T/F)
(c) The assignment symbol used in Pascal assignment statements is =.(T/F)
(d) Values output by **write** statement are not separated by any space.(T/F)
(e) The width of a number, to be output by a **write** statement, can be controlled.(T/F)
(f) The accuracy, upto which a real number can be represented in computer memory, does not depend on the computer word size.(T/F).
(g) The declaration part of a program can follow the execution part.(T/F)
(h) The **writeln** statement can be used to control the vertical spacing of the program output.(T/F)
(i) Lables in a program need not be unique.(T/F)
(j) The semicolon acts as a separator between statements.(T/F).
(k) Transfer to any statement within a compound statement is allowed. (T/F).

5.2. *Complete the following sentences with correct words.*

(a) The first statement of a Pascal program must always be ....
(b) The execution part of a program is always enclosed between the keywords .... and ....
(c) All .... in statements/declarations are ignored by the Pascal compiler.
(d) The label of a statement must be an .... number and is separated from the statement by a ....
(e) The **const** declarations always come .... the **var** declarations.

(f) The constant identifier is separated from the constant value by the sign ....
(g) The compound statement is always designed using the keywods ....
and .... and does not contain any ....

5.3. Explain the ways of intializing a variable in a program. What happens when an assignment statement is executed?

5.4. Look at the following declarations:

```
var
   A, B, C : real;
   X, Y, Z : boolean;
   J, K, L : integer;
```

(a) Indicate the values assumed by the variables in the following assignment statements:

```
A := 3 + 10 * 2;
J := A/2.0;
K := 4 + (7-5) * 3 ;
B := 9 div 2 + 15 mod 6 - J ;
X := J > K ;
Y := X and ( A < B ) ;
C := A+B - J div K ;
Z := not ( Y and X) or ( ord ('B') > ord ('A') ) ;
L := pred (J) - succ (K) ;
```

(b) Indicate which of the following statements are in error. Correct them.

```
X := K+L ;
A := 7/2 + abs (J-K) ;
Y := Z and L
K := A > Z ;
X := (A < B) and C ;
X := (2 * X = Y) ;
```

5.5. How are comments indicated in a Pascal program? Explain the utility of having comments in a program.

5.6. Explain the functions of the **write** and **writeln** statements in a program. Write their formats and illustrate by examples.

5.7. Give the values and organization of the output obtained with the following statements.

```
  (i) write (17 div 2, sqrt (abs (18 -9*6.0)) ) ;
 (ii) write (16*8 : 5, 3/7 : 8 : 2) ;
(iii) write (9, 15, 20.4) ;
      write (-8, 7 * 5);
 (iv) writeln  (output. 'ANSWER =', pred (36)/7.0) ;
  (v) writeln ;
      write  (5.5 : 6 : 2, 19 : 6) ;
      write (15, 28) ;
      writeln ;
      write (- 35 mod 4);
```

5.8. Prepare a program to convert the teperature 37.0° Celcius into its equivalent Fahrenheit.

5.9. Develop a program to find the number of 50 paise, 25 paise, 10 paise, 5 paise and 1 paise coins in the amount Rs.113.99.

5.10. The floor of a room is 15 feet long and 12 feet wide. It is required to fix tiles 4 inches by 3 inches on the floor. Develop an alogirthm and a program which can tell us as to how many tiles are required?

5.11. What do you understand by formatting of output data? Describe the way it is done in Pascal.

5.12. Explain the difference between **read** and **readln** statements. Illustrate by examples.

5.13. Consider the following data on the screen of a display unit:

```
4  9  −3  6  85
```

What valu s are assigned to integer variables J, K, L, M, N by the following statements:

(i) **read** (J, K, L);
(ii) **read** (L, M, N);
(iii) **readln** (J, K, L, M, N);
(iv) **read** (J, K);
readln (L, M, N)

5.14. Describe the structure of declaration part of a Pascal program, giving the function and form of each of the specification.

5.15. Explain the concept of labels and their specification in a program.

5.16. Is it possible to assign names to constant quantities? Give examples. How is the type of the identifier determined?

5.17 Develop a program to compute the mean of 4 real numbers read from the input device. Obtain the hexa representation of its integer part.

5.18. Given the hexa number $(12)_{16}$. Write a program to obtain its octal system representation.

5.19. Develop a program to find the area of a circle. Read the value of radius from the input device. Write the results as

```
RADIUS OF CIRCLE =
AREA OF  CIRCLE =
```

5.20. Modify your program of Exercise 5.9 in such a way that you read data from the input device and prepare the output as

```
THE GIVEN AMOUNT IS Rs. =

NUMBER OF 50-p COINS =
NUMBER OF 25-p COINS =
NUMBER OF 10-p COINS =
NUMBER OF  5-p COINS =
NUMBER OF  1-p COINS =
```

5.21. What do you understand by a compound statement? How is it different from a simple statement? Bring out the advantages/disadvantages which accrue from the facility of having compound statements in a language. Prepare a syntax chart of a compound statement.

5.22. Summarize the complete structure of a Pascal program. Is it useful to have fixed structure of a program? Comment.

Chapter 6

Enumerated and Subrange Data Types

You have learnt about standard data types which are available as part of the Pascal language. They are built-in data types. Pascal also provides a facility for declaring and defining new data types by the user according to the problem and programming needs. Such data types are referred to as Enumerated Data types. Moreover, facility is also available whereby a data type can be created from a subset of the existing integer or enumerated type, which is applicable over a range. Such data types are known as Subrange type. Enumerated and subrange data types extend the typing capability of Pascal enormously and help to develop self-documented programs. They also provide protection from common programming errors and capability for compile and run-time checking of the program execution. We shall study these new data types now.

6.1 Enumerated Data Types

An enumerated data type is a user-defined data type. For such data types,

- allowed values are specified in a list,
- permitted operators are assignment and relational operators only.

An enumerated data type may be declared as

type
enumerated-type-identifier = (i_1, i_2, i_3. . . . . ,i_n)

where

enumerated-type-identifier	indicate the identifier of enumerated type,
i_1, i_2,. . . . .i_n	are the identifiers which form the *item-list.*

The *item-list* = i_1, i_2, ... , constitutes the allowed values (domain) which the enumerated type variable can assume. These can only be characters or identifiers. The list must be enclosed within parentheses as shown.

Variables of enumerated type can be defined as

var
v_1, v_2 . . . ., v_n : *enumerated-type-identifier*

where $v_1, v_2, \ldots, v_n$ are variable names which are declared as of enumerated type.

Examples of enumerated type and variables are given below:

Suppose we wish to have variables which can assume months of the year as their values. Pascal does not permit such variables as standard, so we have to define these ourself. This can be done as:

```
type
   MONTHS = (JAN, FEB, MAR, APR, MAY, JUNE, JULY, AUG, SEP,
             OCT, NOV, DEC)                                  (i)
```

This defines the type called MONTHS which has 12 elements or items making up its domain. Variables of type MONTHS can be defined as

```
   var
     A, B, P : MONTHS
```

Here A, B, P are enumerated type variables. We can perform the following operations with these variables.

(i) Assignment

```
        A := OCT;
        B := JAN;
```

(ii) Comparison using the relational operator as:

```
        A > B
```

The order in which the items are listed in the domain of an enumerated type, determines the order of the items. As for example, JAN precedes FEB, FEB precedes MAR, and so on. This allows the definition of the relational operators, such as

```
=   >   >=   <   <=   <>
```

and the functions predecessor (**pred**), successor (**succ**) for the enumerated type entities as well.

The Pascal compiler automatically assigns ordinal numbers to the elements of the domain. Thus, in example (i), the ordinal numbers of the various items in the list are

```
   JAN  FEB  MAR .....OCT  NOV  DEC
   1    2    3        10   11   12       ← ordinal number
```

(In some Pascal implementations, numbering may start from 0). Thus

```
      JAN > FEB ⇒ FALSE
      OCT < DEC ⇒ TRUE

      pred (NOV) ⇒ OCT
      pred (APR) ⇒ MAR
      succ (FEB) ⇒ MAR
      succ (DEC) ⇒ undefined
```

Further, examples of enumerated data type and variables are:

```
type
     RAINBOW= (VOILET, INDIGO, BLUE, GREEN, YELLOW,
               ORANGE, RED);
     PLANETS = (EARTH, MOON, JUPITER, SATURN, MARS,
               NEPTUNE);
     SAREE   = (NYLON, COTTON, SILK, SHIFFON);
var
         X, Y : RAINBOW;
      Q, T, H : PLANETS;
  PARTYDRESS: SAREE;
```

Variables X, Y are of type RAINBOW and can assume value from the list: (VIOLET, INDIGO, BLUE, GREEN, YELLOW, ORANGE, RED). Similarly, variables Q, T, H are of type PLANETS and they can be assigned values from the list: (EARTH, MOON, JUPITER, SATURN, MARS, NEPTUNE), whereas the variable PARTYDRESS is of type SAREE and can assume data items NYLON, COTTON, SILK and SHIFFON as its values.

Variables of enumerated type can also be declared straightaway as:

---

var
  $i_1, i_2, i_3, \ldots$: *(item-list)*

---

where

$i_1, i_2, \ldots$	variable identifiers
item-list	items which define the domain

Examples of declaring enumerated type variables this way are :

```
var
   WEEKDAY: (SUN, MON, TUES, WED, THURS, FRI, SAT);
   VEHICLE: (CYCLE, SCOOTER, VICKY, CAR, BUS, TRUCK);
```

The variable WEEKDAY can be assigned any value from the list SUN, MON, TUES, WED, THR, FRI, SAT. Similarly, for variable, VEHICLE, any element from its domain can be assigned to it.

The same domain of items can belong to more than one variable. Thus, for the declaration,

```
var
 COLDRINK, HOTDRINK:(CAMPA, LIMCA, TEA, COFFEE, JUICE);
```

variables COLDDRINK, HOTDRINK have the same domain. You can assign values as :

```
COLDDRINK := CAMPA;
 HOTDRINK := COFFEE;
```

Enumerated type variables cannot be used in **read/write** statements. They can only be initialized by the assignment statment. As for example, it is incorrect to state

```
readln  (HOTDRINK);
writeln  (COLDDRINK);
```

The same item should not belong to the domain of two enumerated type variables. For instance, it is wrong to indicate:

```
var
    BOOKS : (ENGLISH, PUNJABI, HINDI, URDU, SINDHI) ;
    SUBJECTS : (ENGLISH, PHYSICS, CHEMISTRY, MATHS) ;
```

because the item ENGLISH appears in the list of both variables BOOKS and SUBJECTS.

The standard type *boolean* is also an enumerated type, defined as

```
type
  boolean = (false, true);
```

where **false** < **true.** As boolean is a standard type, so there is no need to define it.

The function **ord** is defined for the enumerated data types. It gives the place of the item in the domain, starting with 1 (or 0 depending on the Pascal implementation). Example 6.1 illustrates this :

Example 6.1

Define an enumerated type variable, PARTICLE, as:

```
PARTICLE : (ELECTRON, POSITRON, PROTON, NEUTRON,
            PHOTON)
```

Obtain the ordinal numbers of each of the items in the list and compute their sum.

Program 6.1

```
program ORDNUM (output);

   { To compute the ordinal numbers of items of a list in an enumerated type
   variable }

   var
     PARTICLE : (ELECTRON, POSITRON, PROTON, NEUTRON,
                 PHOTON);

     A, B, C, D, E, S : integer;
```

```
begin
   A : = ord (ELECTRON) ;
   B : = ord (POSITRON) ;
   C : = ord (PROTON) ;
   D : = ord (NEUTRON) ;
   E : = ord (PHOTON) ;
   S : = A+B+C+D+E ;
   writeln  (A, B, C, D, E) ;
   writeln  ('SUM =', S)
end.
```

Output

```
   1 2 3 4 5
   SUM = 15
```

This is a simple example and illustrates finding the ordinal numbers of items of enumerated type variables.

The domain of enumerated type identifiers cannot consist of numeric or character constants. For example, the following is not permitted.

```
type
   NUMBER = (1, 3, 5, 7, 9, 11, 13);
```

The reason being that 1, 3, 5, 7, 9, 11, 13 are of integer type. By using numbers in the domain of the enumerated type, we are redefining the type and this is illegal. Similarly, the declaration.

```
type
   KARACTER = ('A', 'B', 'C', 'D', 'E')
```

is not allowed but

```
type
   KARACTER = (A, B, C, D, E)
```

is allowed. See that A, B, C, D, E are not same as 'A', 'B', 'C', 'D', 'E'.

The variables of an enumerated type can be used as selectors in **case** statement (Chapter 7), as control variables in iterative statements (Chapter 8) and to design relational expressions.

The definition and use of enumerated data types is simple and it enhances the expressive power. The programs are better documented and readable.

6.2 Subrange Data Types

Several times, we want that a variable should assume values which lie within a certain range only. For example, consider that the age of adult population of a census data lies between 18 to 100 years. We define a variable as

```
var
   CENSAGE : integer
```

and want that the variable CENSAGE should assume values only from 18 to 100 (both inclusive) and no other value. If any other value, say 101, is assigned, it should immediately be pointed out. This means that the values which the variable CENSAGE can assume are restricted to the range 18 to 100. Such variables are said to be of Subrange type. The values in a subrange are restricted to a set of values. The variable CENSAGE, of subrange type, is declared as

```
var
  CENSAGE : 18 .. 100
```

Pascal allows the definition of both subrange type and subrange variables. The form of subrange type is:

type
type-identifier = *lowerlimit* .. *upperlimit*

The *lowerlimit* and *upperlimit* must be constants, or previously defined constar. identifiers (of type integer) or items of an *enumerated-type-list.* Moreover, *lowerlimit* is always less than the *upperlimit.* The specification *lowerlimit* .. *upperlimit* is referred to as the range of allowed values which a variable of subrange type can assume.

Examples of subrange type declaration are

```
const
  N = 13;
  M = 213;
type
  NUMBER = 1 .. 100 ;
  VALUE  = N .. M ;
  LETTER = 'A' .. 'Z' ;
```

Here NUMBER, VALUE, LETTER define subrange types. Variables of these types may be declared as

```
var
  RESULT, ANSWER : NUMBER;
  X, Y, Z : VALUE;
  W : LETTER
```

This defines variables RESULT, ANSWER of subrange type NUMBER. Values, which these variables can assume, must lie within 1 to 100 (both limits inclusive). Similarly X, Y, Z are subrange type variables with their allowed values restricted to the range defined by the previously defined constants N (=13) and M (=213). Variable W is of type LETTER and can assume any value from the list : 'A', 'B', ... , 'Z'.

Subrange variables can also be defined directly as

```
var
  TEMPERATURE : 89..293;
  PRESSURE    : 1..1000;
  OHM, VOLT   : 1..25;
```

Subrange type can be created from a subset of an already defined enumerated type as well.

The following example illustrates this :

```
type
  MONTHS           = (JAN, FEB, MAR, APR, MAY, JUN, JUL,
                      AUG, SEP, OCT, NOV, DEC);
  SUMMERMONTHS     = MAY..AUG;
  SPRINGMONTHS     = FEB..APR;
  RAINYMONTHS      = JUL..AUG;
```

But declaration of the type

```
     WINTERMONTHS = NOV..JAN;
```

is not allowed, because NOV > JAN (?).

The declaration

```
var
  XX, YY : SUMMERMONTHS;
```

defines variables XX and YY as of type SUMMERMONTHS. These variables can assume values from the list : MAY, JUN, JULY, AUG.

We have seen that a variable of subrange type can assume integer, character or enumerated type of values. When we declare

```
var
  NUMBER : -100..100;
```

the values of variable NUMBER must lie between the limits —100 to 100 .The values are of type integer, though range is —100 to 100. Thus, the host data type is **integer.** Similarly, in the example

```
type
  SYMBOL = 'A'..'P';
```

the host is of character type.

Example 6.2

Refer to Example 5.3. Rewrite this program using the subrange data type for the integer variables.

Program 6.2

We rewrite the declaration part only

```
program ODDEVEN (input, output);
```

```
var
  W : boolean;
  N : 0 .. 15;
  Q1, Q2, Q3, Q4 : 0 .. 7 ;
  R1, R2, R3, R4  : 0 .. 1 ;
begin

  { Execution part is as given in Program 5.3 }
end.
```

Let us study the declaration parts of Program 5.3 and Program 6.2. In Program 5.3, variables N, Q1, Q3, Q4, R1, R2, R3, R4 were of *integer* type and could assume any integer value. Suppose, we give the value 23 to N. The program will run but the result obtained will be wrong. Thus, the correctness of the result is lost. The user has to keep a watch on the correctness of the input data himself or use other statements (such as control statements) to ensure the accuracy of the input values. The disadvatage of using more statements here would be that the program needs more memory and execution time and hence tends to be inefficient. An alternative way to build in the check on the values of the variables N, Q1, . ., Q4, R1, . ., R4 is by declaring them of subrange type, as done in the declaration part of Program 6.2. Now,if the value of N lies outside the range 0 .. 15, an error message will appear and the program will not run. Q1, . . ., Q4 can assume values between 0-7 and R1, . . ., R4 can take values 0 or 1.

The operators and functions defined for the host data type items are also applicable to the subrange type items. Moreover, the result produced by applying an operation to operands of a subrange type may or may not lie in the same subrange. It may belong to the host range. Consider the following example:

```
program PROD (output);
  type
    X = 1 .. 100;
  var
    J, K, L, M : X;
    N : integer;

  begin
    readln (J, K, L);
    N := J* K* L;
    writeln (N: 4)
  end.
```

Here variables J, K, L are of subrange type with values lying in the range 1 .. 100. Suppose, values read for J, K, L are as: 10, 15, 20. Then N assumes the value 10 × 15 × 20 = 3000. This is allowed because N can assume any integer values. However, if we specify

```
M := J* K* L
```

then M would assume the value 3000, which is not permitted, because values assinged to M must lie in the range 1 .. 100.

The subrange type is not defined for real data (?).

6.3 Uses and Limitations of Subrange Data Types

Subrange type variables offer the following advantages:

- the machine can check that all values, assigned to the subrange type of variables, lie within the given range,
- enable to save computer memory,
- program is better documented,
- help to develop a correct program.

As mentioned in the introduction of this chapter that the availability of enumerated and subrange data types are quite useful features. They help us to express a program in a more natural way, better documented, and provide more information to the compiler as well. This also yields advantages of transparency, security and efficiency of program, the features we had mentioned in Chapter 2.

Recall that the transparency of a program implies the clarity and ease with which we can understand a program and follow its execution. This gives us more confidence as to what the program is doing, how it is doing, and whether it is doing what we want it to do. Debugging of programs also becomes easier.

Security of a program concerns the detection of errors by the computer. Errors may occur during the compilation as well as execution phase. It is much easier and economical to correct errors detected at compile-time, than at the run-time. Subrange data type help to locate more errors at the compile time, thus enhancing the program security.

Let us again consider the statement

```
M := J* K* L
```

Here M is a variable of subrange type. During execution, attempt will be made to assign the value 3000 to M. The compiler can not detect this subrange violation at compile time. Thus, there will always be subrange violations which cannot be detected until run-time. This, no doubt, reduces the program security.

Suppose we make the assignments

```
J := 135 ;
K := −80 ;
```

Syntactically, these are correct statements and some Pascal compilers may not point out these as subrange errors at compile time, but good compilers should report this. Similarly, the assignment

```
J := K−M
```

may be legal or not, depending on the values of K and M. It may be legal as long as K $\geqslant$ M, otherwise illegal. Occurrence of this type of situations in a program should be avoided as far as possible.

Program efficiency concerns the amount of memory space and time that the program needs for execution. These two requirements are generally contradictory. Programs written to take up less memory, normally need more time and vice versa and so there is often a trade-off between the two. Similarly, there is a trade off between efficiency and security. Run-time checks increase security but increase execution time. Use of enumerated and sub-range types can help to increase the security of a program without affecting its efficiency. In fact, in some situations, it has been observed that efficiency improves. However, remember that efficiency is a system dependent quantity and may vary from one compiler to another.

Exercises 6

6.1. Tick the correct answers:

(a) Enumerated data types are standard/user-defined data types.
(b) Values for enumerated variables can/cannot be read with **read** statements.
(c) The standard boolean type is/not an enumerated type.
(d) The lower-limit of a subrange variable can/cannot be greater than the upper limit.
(e) The subrange type can/cannot be of enumerated type.
(f) The operators defined for the host data type are/not applicable to the subrange type items.
(g) The subrange type can/cannot be of real type.
(h) The domain of enumerated type can/cannot be of numeric type.

6.2. Complete the following sentences :

(i) Subrange data type can be defined from the existing .... or .... type.
(ii) .... numbers are associated with items appearing in the enumerated list.
(iii) Enumerated type variables can only be initialized by the .... statements.
(iv) The domain of enumerated type identifers cannot be of .... type.
(v) The subrange type identifier must have .... limits.
(vi) Program is better documented with .... and .... type identifiers.
(vii) The items appearing in the list constitute the .... of enumerated type identifiers.

6.3. Explain the concept of enumerated data type. Give examples and illustrate their utility in program development.

6.4. Define data types and variables which can take on the values:

(i) BEER, RUM, WINE, WHISKY, GIN
(ii) DELHI, BOMBAY, MADRAS, CALCUTTA, CHANDIGARH, KASHMIR
(iii) MORNING, NOON, EVENING, NIGHT, MIDNIGHT
(iv) STATEBANK, CANARABANK, BANKOFINDIA, GRINDLASYBANK, BANKOFBARODA, RESERVEBANK, PNBANK

6.5. Determine which of the following are in error. Correct them after giving reasons for errors.

```
type
    FAMILY = (FATHER, MOTHER, SON, DAUGHTER, GRANDFATHER,
    DIRECTION = (EAST, WEST, NORTH, SOUTH);
    INDEX = (-13..113);
    NUM = (7, 8, 9..15);
```

```
var
    M, MEMBER : FAMILY ;
    HIGHWAY   : SOUTH;
    MASS      : 77..50 ;
    X, Y      : BOOLEAN;
begin
    ..........
    readln (M) ; writeln (X) ;
      MEMBER  : = 21;
            M.  : = pred (13) ;
    HIGHWAY   : = NORTH;
    ..........
    ..........
    X : = succ (SON) ÷ pred (13);
    Y : = ord (EAST) – ord (SOUTH);
```

6.6. Refer to Exercise 6.4. Write a program to display the ordinal numbers of the items of the various domains.

6.7. What operators and functions are defined for enumerated data types? List them and explain their use using the data of Exercise 6.4.

6.8. Describe the design and use of subrange data types. What operations are allowed on them?

6.9. Define the following:

(i) Subrange type for the numerals 20 to 35.
(ii) data type for the animals (both pet and wild) and then subrange type for pet and wild animals.
(iii) enumerated type for the names of television sets (both black and colour) and then subrange type for coloured sets.
(iv) data type which can assume the values as AND, OR, NOT, XOR, NAND, NOR and then as NOT, NAND and NOR alone.

6.10. Read a positive integer number lying between the limits 0 to 100. Write a program to convert it into octal system. Print your results with proper headings.

6.11. Look at the following example :

```
program TEST (output);
  var
  A : 20 .. 30;
  B : 40 .. 60;
  C : 80 .. 120;
  D : integer ;
begin
  A: = 2*3+6;
  B: = sqrt (100 * A);
  C: = sqrt (100 * A);
  D: = A * B * D;
  writeln (A, B, C, D)
end.
```

Run this program on your system. What do you get as the output? Are the results correct? Analyze the output and make the program run.

6.12. Indicate

```
pred (false) =
succ (true) =
ord (false) – ord (true) =
```

Chapter 7

Program Execution Control

Normally, a computer executes statements of a program sequentially, that is, one after another in the serial order. In actual problems, this is seldom the case and the sequence of statement execution may be required to be changed due to several factors and conditions of the solution. The order of statement execution can be controlled by means of various control commands. These commands help to jump from one part of the program to another. This transfer of control may be based on certain conditions or may be unconditional. We shall discuss the control statements which help to do so and the related concepts in this chapter.

7.1 The if Statement

The **if** statement is the simplest form of control statement. It is very frequently used in decision making and altering the flow of program execution. A simple form of an **if** statement is

if *test-condition* **then** S

where

test-condition	a boolean/relational expression
S	a statement, may be simple or compound

We shall refer to this form of **if** statement as **if-then** form. The **if-then** statement operates as follows: When the *test-condition* is true, then the statement S is executed, but when the test-condition is false, statement S is skipped, and the statement immediately following **if** statement is executed. Schematically, we can show the action of the **if-then** statement as given in Fig. 7.1.

Introduce the declarations

```
    var
        A, B, C, D, T, X, Y : real;
                    P, Q, U : boolean;
                       M, N : integer;
      ALPHABET, LETTER : char
```

An example of **if-then** statement is

```
if A < 10.0 then X := 3.0 + sqrt (9.0 * 4.0) ;
X:= A * 13.0;
```

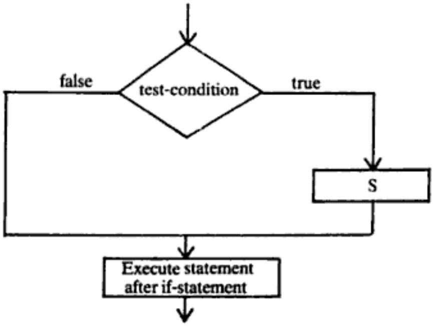


Fig. 7.1: Action of **if-then** statement

This **if** statement specifies that when the test-condition A < 10.0 is true, then execute the statement X : = 3.0 + **sqrt** (9.0*4.0), otherwise, the control of program execution should go to the statement immediately following it, that is, X : = A * 13.0.

Further examples of **if-then** statements are

```
if A+B*C = D * sin (T) then writeln (X, Y) ;
if Q or P then read (A, B, C, D) ;
if (M > N) and (X+Y < A) then N := N+1 ;
```

The statement S may be a simple or compound statement. We can specify an **if-then** statement as

```
if X <> Y then
  begin
    if A = B then C := A+B−C
  end ;
```

The test-condition in an **if** statement may be formed with enumerated or character data as well. Consider

```
type
  WEEK = (MON, TUES, WED, THRS, FRI, SAT, SUN);
  WEEKDAYS = (MON .. FRI) ;
var
  DAY: WEEKDAYS;
  RATE : real ;
begin
  if DAY = SUN then RATE := 50.0;
  if DAY <> SUN then RATE := 20.0;
  ..........
  ..........
end ;
```

Here, WEEKDAYS is an enumerated subrange date type and DAY is a variable of this type. In the first **if-then** statement, when the content of variable DAY is SUN, then rate is initialized to value 50.0, otherwise control goes to the next **if-then** statement. Here, if the value of variable DAY is < > SUN, value 20.0 is assigned to the variable RATE.

Example 7.1

Read a real number. Evaluate and write its square root. If the number is negative, do not perform any computations but print the message that the number is negative.

Program 7.1

```
program POSNUM (input, output) ;
  { To find the square root of a number }
var
  A, B : real ;
begin
  readln (A) ;
  if A < 0.0 then writeln ('Number is Negative') ;
  if A > = 0.0 then
    begin
      B : = sqrt (A) ;
      writeln ('Sqr. Root =', B : 5 : 2)
    end
end.
```

Sample input

```
    -7.0
```

Output

```
        Number is Negative
```

Let us go back to *Program* 7.1 and study it more carefully. Two **if-then** statements have been used and the test-conditions have to be evaluated twice. Now the test-condition A < 0.0 is automatically true if A >= 0.0 is false. If we could have this provision, our program will become more efficient. Pascal allows to handle this type of situations by another more general form of **if** statement. Its form is

if *test-condition* **then** S_1 **else** S_2

where S_1 and S_2 are other statements. Statement S_1 must not be followed by a semicolon. We shall refer to this form of **if** statement as **if-then-else** form. It operates as follows: when the *test-condition* is true, then statement S_1 is executed and S_2 is skipped; on the otherhand, when the test condition is false, then S_1 is bypassed and statement S_2 is executed. Schematically, we can represent this as shown in Fig. 7.2.

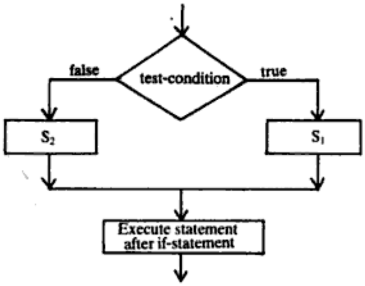


Fig. 7.2: Action of if-then-else statement

Execution control always passes on to the statement after the **if-then-else** statement whether S_1 or S_2 is executed. Statement S_1 and S_2 may be simple or compound. S_1 should never be followed by a semicolon, while S_2 should be.

An example of **if-then-else** statement is

```
if A+B < > C* 4.0 then D := A+B+C
   else D := A − B − C ;
```

This statement instructs the computer to check if A+B and C*4.0 are unequal. If they are not equal, then the variable D should be set equal to A+B+C, otherwise, D should be initialized to the value of the expression A−B−C.

Now look at

```
if P and Q then U := true
  else writeln (P, Q) ;
```

According to this statement, U is assigned the value **true** when the boolean expression P **and** Q is **true**. If this expression is **false**, then the statement **writeln** (P, Q) is executed.

Let us rewrite the execution part of Program 7.1 using the **if-then-else** statement. This becomes

```
begin
   readln (A);
   if A < 0.0 then writeln ('Number is Negative')
                 else begin
                      B := sqrt (A) ;    writeln ('Sqr. Root =', B : 5 : 2)
                      end
end.
```

The reader should appreciate that the use of **if-then-else** statement makes the program more compact, better readable and efficient.

Next, refer to Program 5.3. We had put the condition that N should lie in the range 0-15. One technique of ensuring this is by the use of subrange data type (Program 6.1) and the other is by the use of **if** statement :

```
if N >= 0 or N <= 15 then.........
```

Thus, limits and type of values of a variable can be examined/controlled in two ways: use of subrange type and control statements. Subrange type technique is more efficient as it is used in the declaration part of the program and checked during compilation phase, whereas the control statement technique is used in the execution part of the program and is operative during the execution phase of the program. However, there is one limitation. Subrange type technique cannot be used with real data type, while the control statement can be used for any data types.

The *test-condition* may be designed with arithmetic, boolean, character or user-defined data items as discussed earlier. For instance, in the following statement:

```
if (LETTER < 'A') or (LETTER > 'Z')
  then ALPHABET := false
      else
        ALPHABET := true ;
```

the *test-condition* is based on the use of character data. Similarly, we can write as

```
if (DAY = SAT) or (DAY = SUN) then RATE := 50.0
                 else RATE: = 20.0;
```

Now the *test-condition* has been designed using enumerated data types.

The **if-then-else** statement also enables us to specify multiple actions in a single instruction. As for instance, when this statement is written as

if *test-condition*-1 **then** S_1

  else (if *test-condition*-2 **then** S_2

            else S_3)

then two test-conditions are being examined. Schematically, we can show this as given in Fig. 7.3.

We shall call this as a nested **if** statement. Example 7.2 illustrates the use of nested **if** statements.

Example 7.2

Develop a program to compute the real roots of the quadratic equation

$$ax^2 + bx + c = 0 \qquad (1)$$

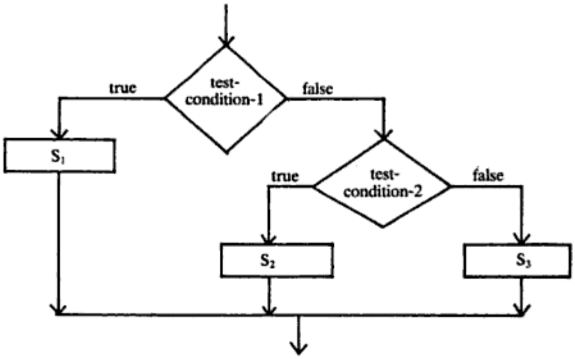


Fig. 7.3: Nested **if** statement

The program should be such that it accepts only nonzero positive values of coefficients, a, b, c.

As mentioned earlier, the roots of the quadratic equation (1) are given by

$$x_{1,2} = (-b \pm \sqrt{(b^2-4ac})/2a$$

(b^2-4ac) is the discriminant of the equation. If $(b^2-4ac) \geqslant 0$, the roots are real. Program 7.2, given below, takes care of all the above restrictions imposed on the values of a, b, c, and the nature of roots.

Program 7.2

```
program QUAD (input, output) ;
  const
       TWO = 2.0 ;

  var
    A, B, C, D, DISC, R, ROOT1, ROOT2 : real ;

  begin
    readln (A, B, C);
    { Check the validity of values of A, B, C }
      if (A = 0.0) and (B = 0.0) and (C = 0.0)
      then writeln ('A =', A, 'B =', B, 'C =', C)
        else if (A < 0.0) and (B < 0.0) and (C < 0.0)
               then writeln (A, B, C)
                    else
                        begin
                          D := TWO* A ;
                        DISC := B*B-4.0*A*C
                        end ;
        if (DISC >= 0.0) then
                begin
                  R := sqrt (DISC) ;
                  ROOT1 := (-B+R)/D ;
                  ROOT2 := (-B-R)/D ;
                    writeln ('A =', A : 4 : 1, 'B =', B : 4 : 1, 'C =', C : 4 : 1)
                    writeln ('Root 1 =', ROOT1 : 6 : 2, 'Root2=', ROOT2 : 6 : 2)
                  end
    end.
```

Sample input

```
-2.0   3.0   6.0
```

Output

```
A=-2.0  B= 3.0  C= 6.0

Root1 =ƀ-1.13  Root2 =ƀƀ2.63
```

Suppose we wish to repeat the execution of program 7.2 for more sets of data of the variables A, B, C. The program, as developed above, cannot do so. For being able to do so, control must be transfered back to the beginning at the statement **readln** (A,B,C,). We have not learnt how to do this so far. The **if** statements do not permit jumping from one part of the program to another. Moreover, program execution flow always goes in the forward direction with their use. Transfer of program execution control from one part to another is possible with the use of **goto** statements. This is described in Section 7.3.

Example Program 7.2 illustrates the use of nested **if-then-else** statements. Though the program is more compact, but transparency may be effected if any further nesting is carried out.

Pascal allows nesting to any level, however, it may depend on Pascal implementation on a particular system. Experience shows that most commonly used forms of **if** statement are **if-then** and simple forms of **if-then-else.** Only in situations where multiple decisions are required, the more complex forms of **if-then-else** statement are useful. However, most of the programs may be written with the help of simple forms of **if** statements. Such programs are easy to understand and debug.

7.2 The case Statement

We have seen that **if** statement enables us to design multiple-way decisions in a program. This can also be achieved by another statement, called **case** statement. The format of this statement is

```
case expr of
   l₁ : S₁ ;
   l₂ : S₂ ;
   ......
   ......
   lₙ : Sₙ
end;
```

where

$l_1, l_2, \ldots l_n$	labels, referred to as **case** labels
$S_1, S_2, \ldots S_n$	Pascal statements
expr	an expression and is known as **case** selector or **case** index.

The **case** labels are different from the labels of statements as defined in Pascal (Section 5.8). **case** labels may be integers, characters, boolean or enumerated data items. Moreover, each of the statements $S_1, S_2, \ldots, S_n$, may have one or more than one **case** label associated with it. As for example,

$l_1, l_2 : S_1$;

or

$l_5, l_7, l_1 : S_2$;

The **case** index may assume values which correspond to the **case** labels. The **case** statement always has an **end** associated with it.

The **case** statement operates as follows: first the value of the **case** expression, that is, the selector is obtained and compared with each of the **case** labels. The statement whose label matches with the selector is executed and then control goes to the statement immediately following the **case end.** When no match occurs, program execution control is transferred to the statement after the **case end.** (Remember, each **case** statement must have its associated **end** word). This action of the **case** can be represented schematically as shown in Fig. 7.4.

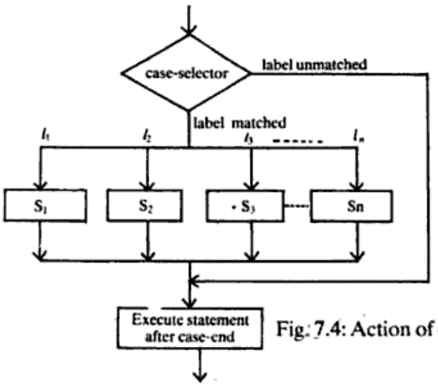


Fig. 7.4: Action of case statement

An example of **case** statement is

```
case J of
  1 : K:= K+1 ;
  2 : K:= K+2 ;
  3 : K:= K+3 ;
  4 : K:= K+4
end;
```

According to this **case** statement, statement labelled 1 is executed when J = 1 and statement 2 is executed for J = 2. When J = 3 and 4, statements labelled 3 and 4 respectively, are executed.

The **case** labels must be distinct. For example, it is wrong to write

```
2 : K:= K+2 ;
2 : K:= K+3 ;
```

as then the machine does not know where to transfer the execution control.

The case-selector may be integer, boolean, character, subrange or enumerated type, but not of real type. This is illustrated by the following examples.

```
type
  FIGURE = (SQUARE,CUBE, RECTANGLE, TRIANGLE) ;
var
  GEOFIGURE : FIGURE ;
  LENGTH, BREADTH, AREA: real ;
  S, A, B, C : real ;
  ........
  ........
  ........
case GEOFIGURE of
  SQUARE : AREA : = LENGTH * LENGTH;                    (i)
  RECTANGLE : AREA : = LENGTH * BREADTH;                (ii)
  CUBE : AREA : = LENGTH * LENGTH * LENGTH;             (iii)
  TRIANGLE : AREA : = sqrt (S* (S-A)* (S-B)* (S-C) )    (iv)
end;
```

Here the variable GEOFIGURE is the case-index or selector. GEOFIGURE is an enumerated type of variable which can assume the values: SQUARE, CUBE, RECTANGLE and TRIANGLE. When its value is CUBE, the statement identified as (iii) is executed. When the value happens to be TRIANGLE statement (iv) is executed, and then control goes to the statement following **end.**

The type of selector-variable and the label must match. As for example, in the above **case** statement, the value of selector-variable GEOFIGURE is of enumerated type and all the **case** labels are also of the same type.

The reader may note that the **if-then-else** statement is a special case of **case** statement as shown below.
Consider

```
    if test-condition then S1 else S2 ;
```

This is equivalent to

```
    case test-condition of
        true  : S1 ;
        false : S2
    end ;
```

Example 7.3 illustrates the use of a **case** statement in an actual program. You know that there are four arithmetic operations: +, −, *, / (division). Any of these operations can be selected in a program with a **case** statement by choosing the case-index to be of character type. One way of writing the program is given in example 7.3.

Example 7.3

Design a program which simulates the four arithmetic operations for real numbers.

Let A and B be two real variables. The four arithmetic operations performed on these are A+B, A−B, A*B, A/B. The program which we write below selects the appropriate expression using **case** statement.

Program 7.3

```
program ARITHOP (input, Output);

var
    A, B, C: real;
    OPR : char;

begin
     writeln ('Enter the two numbers') ;
     readln (A, B) ;
     writeln ('Specify the operator') ;
     readln (OPR) ;
     case OPR of
     '+' : C : = A+B ;
     '*' : C : = A*B ;
     '−' : C : = A−B ;
     '/' : C : = A/B
     end ; { case }
     writeln ('The answer is') ;
     writeln (C : 4 : 1)
 end.
```

Sample input/output

```
Enter the two numbers
4·0     2.0
Specify the operator
+
The answer is
6.0
```

We have seen that many **case** labels may appear with a statement within a **case** statement. This allows the execution of the same statement for a variety of values of the **case** expression. The **case** statement is particularly useful when selecting from various options based on the values which are not in sequence, but are random.

Any statement, simple or compound, may be used within the **case** statement. The following example illustrates the case of **if-then-else** statement within a **case** statement. Suppose, we wish to determine the number of days in the month for a particular year. This may be programmed as:

```
var
  MONTH : (JAN, FEB, MAR, APR, MAY, JUN, JULY, AUG, SEPT,
                                          OCT, NOV, DEC);
  YEAR, DAYS: integer;
      .
      .
      .
case MONTH of
  JAN, MAR, MAY, JUL, AUG, OCT, DEC : DAYS := 31 ;
  APR, JUN, SEP, NOV: DAYS = 30 ;
  FEB: begin
         if (YEAR mod 4) = 0 and (YEAR mod 100 <> 0)
         then DAYS := 29
              else DAYS := 28
         end
end; { of case }
writeln (DAYS);
```

7.3 The goto Statement

This statement causes the transfer of program execution control from one statement to another unconditionally. Its form is

```
goto n
```

where

n label of a statement to which control is to be passed on.

n must be an unsigned integer label whose value can be from 1 to 9999.

Examples of **goto** statement are:

```
goto 20;

goto 113;
```

Labels may be used in **goto** statements which appear before or after this statement. The same label may be specified in any number of **goto** statements.

The **if** and **goto** statements may be used in the compound statement to transfer control out of the compound statement, but it is illegal to transfer control into it. Following examples illustrate this:

```
    . . . .
    . . . .
(a) begin
    . . . .
    . . . .
    . . . .
    if A > B then goto 100;
    . . . .
    . . . .
    end ;
    100 : D = A+B+C;
    . . . . .
    . . . . .
    . . . . .
```

The specification of form (a) is allowed.

(b) The following form is not permitted

```
    . . . . .
    . . . . .
    . . . . .
    begin
    . . . . .
    . . . . .
    . . . . .
    50 : S1;
    . . . . .
    . . . . .
    . . . . .
    end ;
    goto 50
    . . . . .
    . . . . .
```

However, entry into a compound statement can be made at the beginning of the statement. For instance, the following transfer is valid

```
.....
.....
.....
150 : begin
.......
.......
.......
.......
end;
goto 150
.....
.....
.....
```

case label should never be used with the **goto** statements as such labels are distinct from the Pascal labels of statements.

Refer to Program 7.2. As mentioned there, more computations can be repeated for different sets of data values for A, B, C, provided we transfer the program execution control to the beginning of the program. Now this is possible with a **goto** statement. We rewrite the outline of Program 7.2 and introduce the use of **goto** statement as illustrated below.

Program 7.2 (rewritten)

```
program QUAD (input, output);
   label 77;
   const TWO = 2.0 ;
   var
      A, B, C, D, DISC, R, ROOT1, ROOT2 : real ;
                             X : char ;
   begin
      77 : readln (A, B, C) ;
           ...........
           ...........
           ...........
   writeln ;
   writeln ('Do you want to enter more data? (Y/N)')
   readln (X) ;
   if X = 'Y' then goto 77 ;
   writeln ('Stop')
   end.
```

Now the sample input/output may appear as

```
0.0        0.0        0.0
A = 0.0     B = 0.0     C = 0.0
. . . . . . . . . . . . . . . . . . . . . .
```

```
Do you want to enter more data? (Y/N)
Y
1.0     3.0     4.0
. . . . . . . . . . . . . . . . . . . . . .
Do you want to enter more data? (Y/N)
Y
6.0     2.5     3.0
. . . . . . . . . . . . . . . . . . . . . .
Do you want to enter more data? (Y/N)
N
Stop
```

This mode of program execution is referred to as Interactive mode. Another technique of developing interactive programs is described in Chapter 8.

Statements considered so far have been simple and easy to understand. They are adequate to write several interesting and general programs. We illustrate their use further by another example.

We consider the example of generating Fibonacci series. This is a sequence of numbers;

0, 1, 1, 2, 3, 5, 8, 13, ...

Each term is obtained from the sum of the proceeding two terms, that is:

```
0+1 = 1
1+1 = 2
1+2 = 3
2+3 = 5
3+5 = 8
5+8 = 13
8+13 = 21
13+21 = 34
.........
.........
.........
```

This number sequence has several practical applications in areas such as electrical network theory, biological systems, etc. The following program has been developed using the **if** and **goto** control statements. This program can be developed using iterative constructs (Chapter 8) and recursively (Chapter 10) as well. This we will do later on.

Example 7.4

Develop a program to generate the Fibonacci series using the control statements only.

Program 7.4 has been written following the above process of Fibonacci sequence generation.

Program 7.4

```
program FIB (input, output) ;

  { Calculation of Fibonacci sequence using control statement only }
  label 20 ;
  var I, J, K, L, N : integer ;
  begin
    { Initialize variables }
    readln (N) ; { N = No. of terms in the series }
    writeln ;
    K := 0 ; writeln (K) ; I := 1 ; writeln (I) ;
    K : = K+1 ; J : = 1 ; writeln (J) ;
    { I, J, L, give successive terms. }
    20 : K := K + 1 ; L := I+J ; writeln (L) ;
        K : = K+1 ; I : = J ; J : = L ;
  if K < N then goto 20 ;
  { K keeps count of the no. of terms }
  writeln ('The above are the', N, 'terms of the Fibonacci sequence')
  end.
```

Sample input/output

```
    6

    0
    1
    1
    2
    3
    5
The above are the 6 terms of Fibonacci sequence
```

We have discussed in detail the format and use of control statements as available in standard Pascal. A variety of programs, simple as well complex, can be developed using them. The **if-then-else** and **case** statements are very powerful and form the basis of structured design of computer programs. Decisions, based on logical reasoning, can be easily built into a program using **if-then-else** statements. Appropriate nesting of these statements enables to design quite versatile programs. **if** statements can also be used to perform repetitive computations in a program, as you have seen. However, such computations can be better implemented using Pascal's iterative constructs which are discussed in the following chapter.

Exercises 7

7.1. Tick the correct answer:

(a) Statements in a program are normally executed sequentially/randomly.
(b) The test-condition in an **if** statement can/cannot be designed using enumerated data types.
(c) **if** statements can/cannot be nested.
(d) The type of **case** index must/need not match the **case** label.
(e) The **case** selector can/cannot be of real type.
(f) Every **case** statement must/need not have its own **end** word.
(g) Same/different labels must be used with statements in a program.
(h) A compound statement may/may not appear within the **case** statement.

7.2. Complete the following sentences:

(i) The order of execution of statements in a program can be controlled by the . . . . statements.
(ii) The test-condition appearing in the **if** statement may be designed with . . . . or . . . . or . . . . .or . . . . data items.
(iii) Multiple decisions in a program may be easily implemented using . . . . statement
(iv) **case** labels must not be of type . . . .
(v) The statement S_1 appearing after the keyword **then** in the **if-then-else** statement may be . . . . or . . . .
(vi) The action of **if-then-else** statement can be simulated by the . . . . statement.
(vii) Enumerated data items can be used as labels in a . . . . statement.

7.3. Write the different constructs of **if** statements available in Pascal. Draw their syntax charts. Explain their action by examples.

7.4. Assume the following types of variables:

```
var
    A,B,C,D : real;
        P,Q: boolean;
        J,K : integer;
```

Determine errors, if any, in each of the following statements. Correct them.

```
(i) if A > B then A:='true ;
(ii) if P and Q then A:=2.0*sqrt (B);
        else A = 3.0* trunc (B);
(iii) if A+B+C < 100.0 then else B=C/D
(iv) if C = D then
        10 : begin
                A:= 50;
                B: =6.0
            end
        else
            20: begin
                    A:= 10.0;
                    B:= 13.0
                end;
(v) case J of
    'A' : K:= K+1;
    'B' : K:= K+3;
    'C' : K:= K+5      end;
```

7.5. Multiple-branch decision in a program may be coded either by **if** or **case** statements. When and which option would you prefer? Illustrate by examples.

7.6. Explain the action of a **case** statement. Write its format and give examples.

7.7. How are **case** labels different from statement labels? Is it possible to refer to the **case** labels by a **goto** statement? Explain your answer.

7.8. Look at the following two **case** statements:

```
(i) case J of
        4,6 :;
        7,9 :K:=J div 2
        end ;
```

and

```
(ii) case J of
        4,6:
        7,9:K:=J div 2
        end;
```

which of these has the correct format? What can be the possible values of K?

7.9. Generate the action of **if-then-else** statement using **goto** and **if-then** statements.

7.10. The series

$$1 - 1/3 + 1/5 - 1/7 + \ldots\ldots \pm 1/2n+1$$

converges to the value of $\pi/4$. Develop a program to obtain the value of π from this series. How many terms are needed to obtain the value as 3.1416.

7.11. Develop a program to read a character. If it is a letter, output should appear as

IT IS A CHARACTER

If the character is a digit, message should be

IT IS A DIGIT

while if the character is some other symbol, then display

IT IS A SPECIAL CHARACTER

7.12. Develop a program to determine whether the given integer number represents a leap year. (Note: the integer number must be positive).

7.13 Prepare a program to compute the sum of prime numbers lying between two numbers, say, 0 and 200.

7.14. Develop a program, using **case** statement, which evaluates the function f(x) (x is an integer)

$$\begin{aligned} f(x) &= 1+x+x^2 & 0 \leqslant x < 4 \\ &= 1-x+x^2 & 4 \leqslant x < 7 \\ &= 1+x-x^2 & 7 \leqslant x < 10 \end{aligned}$$

7.15. Design a program to compute and print n! (factorial of n) as long as $n! \leqslant 10^5$. Print the value of n for which this limit is reached.

7.16. Develop a flowchart and a program to locate the largest positive integer such that

$$1^3+2^3+3^3+4^3+\ldots\ldots+n^3 \leqslant 10^5$$

7.17. Prepare an algorithm and a program which reads two positive numbers m and n and prints all powers of n which are less than m.

7.18. Develop a program to compute the value of n for the sum

$$\sum_{n=0}^{?} 1/n!$$

to converge to 2.71828.

7.19. Develop a program which converts decimal integers into their equivalent binary, octal and hexadecimal representation using **case** statement.

7.20. A computer manual has the following information about the diagonistics issued by the system:

Code	Error
0	All correct
1	invalid input
2	data values incomplete
3,4	wrong syntax
5,6,7	division by zero
8,9	error undefined

Prepare two programs using (i) **case** and (ii) **if-then-else** statements. Which program is easier to write and efficient to execute?

7.21. Define a data type and variable which can take on the values FIAT, AMBASSADOR, MONTANA, STANDARD, MARUTI, CONTESSA, NE118. Write a program to ask a person to reply YES or NO to questions indicating which car he is thinking of?

7.22. Develop a program to prepare the truth tables of NOT, OR, AND logical operators.

Chapter 8

Repetitive Computations

Often in our computations, we need to repeat a set of calculations many times. This can be done with the help of **if** statements. However, there are other ways also by which repetitive computations (looping) may be performed in a program. This is with the help of **for-do, while-do** and **repeat-until** statements. They are also referred to as Looping or Iterative Constructs. These statements enable us to make best use of structured data type — such as arrays — in varied applications like iterative computing, matrix manipulations, large scale data processing, business applications, and so on. We shall discuss these statements, their use and applications here.

8.1 The for-do Statement

The format of this statement is

$$\textbf{for } v := e_1 \begin{bmatrix} \textbf{to} \\ \textbf{downto} \end{bmatrix} e_2 \textbf{ do } S$$

where

v	variable of ordinal type,
e_1, e_2	expressions,
S	statement, may be simple or compound

Variable v is called the Control or Index variable. The expression e_1 represents the initial value of v while e_2 is the final value which the control variable can assume. These expressions must be of ordinal type (that is, integer, boolean, character, subrange or enumerated) and their type must match the type of control variable. The symbol [ ] indicates that either of the options **to** or **down to** is to be specified. Thus,

The **for-do** statement may have the form

for $v := e_1$ **to** e_2 **do** S (i)

or

for $v := e_1$ **downto** e_2 **do** S (ii)

Let us first examine the action of form (i). According to this form of **for** statement, execute statement S, for $v = e_1 = p_1$ (say), $v =$ **succ** $(p_1) = p_2$ (say), v = **succ** $(p_2) = p_3$ (say), till v equals the value e_2. Here $p_1, p_2, p_3, \ldots$ are the values which the variable v can assume. When the value of $v > e_2$, the statement

S is not executed. We can represent this action of **for-do** statement as given in Fig. 8.1.

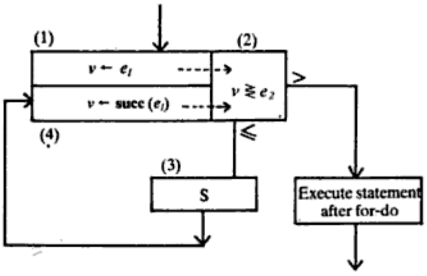


Fig. 8.1: Action of for-do statement

The machine calculates the value of expressions e_1 and e_2 only once in the beginning. The control variable is assigned the value of e_1 [box (1)]. Machine examines whether this value is $\leqslant e_2$. If the value of $v \leqslant e_2$, statement S is executed [box (3)] and *v* is assigned the next (succeeding) value of e_1 [box (4)]. This value is obtained by the use of the function **succ** automatically by the system. The new value of *v* is again compared with the value of e_2 [box (2)]. If $v > e_2$, statement S is not executed and control passes to the statement following the **for** statement, otherwise, the previous process is repeated. This goes on as long as $v \leqslant e_2$.

The syntax chart of the **for-do** statement appears as shown in Fig. 8.2.

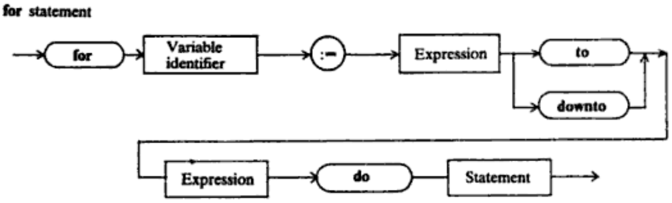


Fig. 8.2: Syntax chart of for-do statement

An example of **for-to-do** statement is

for J : = 1 **to** 10 **do write**(J) ;

Here the control variable is J, $e_1 = 1$ and $e_2 = 10$. According to this statement, execute the statement **write** (J) for J = 1, 2, 3, 4 . . . 10. We will obtain the printed values as 1 2 3 4 5 6 7 8 9 10. Consider

```
for K := M to N do
    J : = M * N + K;
```

The statement J : = M * N + K is executed for K=M, M+1 M+2, . . . , N. If K > N at the start, then statement J : = M * N+K is not executed even once.

The action of form (ii) of **for-do** statement, that is, of

for v: = e_1 **downto** e_2 **do** S

is similar to that of **for-to-do** form, except that now the statement S is executed for $v = e_1 = v_1$ (say), then $v =$ **pred** $(v_1) = v_2, \ldots$, till value of control variable $< e_2$. The statement S is executed for $e_1 \geqslant e_2$.

Examples of **for-downto** statement are

```
for NUMBER := 10 downto 5 do readln (PRICE) ;
for TAX := MAX downto MIN do
  begin
    readln (RATE);
    writeln (RATE * (MAX−MIN)/2.0−TAX)
  end;
```

In the second statement, there is a compound statement following **do.** This compound statement will be executed for TAX = MAX, MAX − 1, MAX − 3, ... till TAX = MIN. Here it is assumed that TAX, MAX, MIN are integer variables.

The following statement illustrates the calculation of sum of odd and even numbers from 1 to 100.

```
for J := 1 to 100 do
   if ((J div 2) * 2−J) = 0
     then EVEN := EVEN+J
       else ODD := ODD+J ;
```

assuming that variables EVEN and ODD had been initialized to zero previously.

Program 8.1

Develop a program to compute the value of natural logarithm base, e, from the relation :

$$e = \sum_n 1/n! \quad n=0, 1, 2, \ldots$$

Here, symbol ! indicates factorial and implies n! = n. (n−1). (n−2) . . . . 1. Also 0! = 1. Print the sum of successive terms for n=0, 1, 2, . . ., 10.

Program 8.1

```
program NATLOGBASE (output) ;
  var E : real ; K, N : integer ;
```

```
    begin
      E := 1 ;{ N =  0 term }
      K := 1 ;
      writeln ('N', 'ƀƀƀƀ', 'Sum') ;
      writeln (N : 3, E : 16 : 6) ;
      for N := 1 to 10 do { loop to add terms }
        begin
          K := K * N ; { factorial is calculated and stored in K }
          E : = E + 1.0/K
          { sum of successive terms is stored in E }
           writeln (N : 3, E : 16 : 6)
        end ;
        writeln ('Value of e = ', E : 10 : 6)
      end.
```

Output

```
  N                Sum
   0               1.000000
   1               2.000000
   2               2.500000
   3               2.666667
   4               2.708333
   5               2.716667
   6               2.718056
   7               2.718254
   8               2.718278
   9               2.718278
  10               2.718278

  Value  of  e  =  2.718278
```

The statement S following the keyword **do** may be simple or compound. S is also referred to as the Body or Range of the **for-do** statement. The values of control variable, expressions e_1 and e_2 must not be modified or redefined in the range of the **for-do** statement. The following example illustrates this:

```
for J : = M to N do
  begin
101 : P : = M+N+J;
      .......
      .......
102 : M : = J+13;
103 : J : = 7;
     .......
     .......
  end;
```

In statement labelled 101, the sum of values of M, N, J is assigned to variable P. The values of variables M, N, J are not modified. This is allowed. However, in statements, labelled 102 and 103, value of M and control variable J are altered. This is not permitted. Thus, the control variable may be used in other statements provided its value is not changed. When the **for-do** loop is completed, the value of control variable is undefined.

It is possible that the control leaves the range of **for-do** because of execution of a **goto** or **if** statement. In that case, the current value of control variable is saved and can be used outside the range of the **for-do** statement. This is made evident by the following illustration.

```
for K := P to T do
   begin
   .......
   .......
     if X > Y then goto 13
   ........
   end;
   13:.......
       .......
       .......
       .......
```

Here the **for-do** statement has a compound statement as its range. When the condition X>Y is satisfied, execution control is transferred to statement labelled 13, outside the range. In such situations, the control variable K will retain the value it assumed before initiating the execution of the compound statement. This value may be used in any other expression/statement as desired.

Further examples of **for-do** statement, using other ordinal data types are :

```
for LETTER := 'A' to 'Z' do
   writeln (LETTER);

for MONTH := JAN to DEC do
   case MONTH of
         JAN, MAR, MAY, JUL, AUG, OCT, DEC : DAYS := 31 ;
         APR, JUN, SEP, NOV : DAYS := 30 ;
         FEB : DAYS := 28
   end;

for LOJIK := false to true do
   if LOJIK and true
      then writeln (LOJIK)
           else B := not LOJIK;
```

According to the last statement, value of variable LOJIK will be written if the expression LOJIK **and true** gives the true value, otherwise the boolean variable B is assigned the value **not** LOJIK.

The **for-do** loops may be nested. When so, the variables, that control the loops must be distinct. An example is

```
for J := 1 to 4 do
  for K := 4 downto 1 do
    SUM := SUM+J*K ;
```

This is referred to as Nesting of loops. Loops may be nested to any extent.

8.2 The while-do Statement

You have seen that the value of the control variable governs the repeated execution of statements in the **for-do** statement. In several applications this mode of repetitive computations may not be convenient and we may desire that the calculations should be carried on till some test-condition is satisfied. This is possible with the **while-do** and **repeat-until** statements. We consider the **while-do** statement here and take up the discussion of **repeat-until** in the next section.

The format of **while-do** statement is

while *test-condition* **do** S

where

test-condition	a boolean or relational expression whose value may be true or false,
S	a statement, may be simple or compound.

The action of the **while-do** statement is as: execute statement S as long as the *test-condition* is true. Schematically, we can show this action as given in Fig. 8.3.

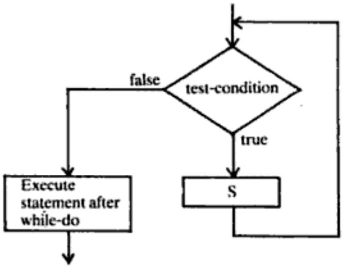


Fig. 8.3: Action of while-do statement

First, the value of the *test-condition* is obtained. If the value is true, S is executed and control goes back. Again the *test-condition* is evaluated and its value tested. If the value is true, S is executed, otherwise control goes to the statement immediately after the **while-do** statement. Thus, the execution of the **while-do** statement is corripleted only *when the test-condition is found to be false.* If the *test-condition* is false at the start, the statement S is not executed even once.

The syntax chart of **while-do** statement appears as given in Fig. 8.4.

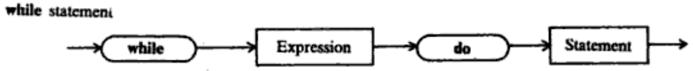


Fig. 8.4: Snytax chart of while-do statement.

An example of **while-do** statement is

```
while X > 0 do
  begin
    SUM: = SUM + X;
    X:=X-2.0            Compound statement
  end;
```

According to this statement, the compound statement is executed as long as the value of X is positive and non-zero. When X becomes less or equal to zero, the compound statement will not be executed.

Other examples of **while-do** statement are.

```
while not eof (input) do ........
while (KARACKTER < > PERIOD) do ........
```

Example 8.2

Read the characters of a sentence one by one. Count the number of vowels. The program should stop when the character fullstop (.) is encountered.

Program 8.2

```
program VOWEL (input, output) ;
  { To count the number of vowels in a sentence }

var
    CH : char ;
    KOUNT : integer ;

begin
    KOUNT := 0 ;
    readln (CH) ;
    while (CH < > '.') do
      begin
        if (CH = 'I') or (CH = 'O') or (CH = 'U') or (CH = 'A') or
                                             (CH = 'E')
```

```
          then
            KOUNT := KOUNT + 1 ;
        readln (CH)
        end ;
    writeln ('The number of vowels in the sentence =', KOUNT)
    end { of the program }.
```

Sample input

```
    C
    O
    M
    E
```

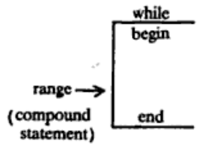
Output

```
The number of vowels in the sentence = 2
```

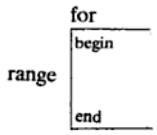
The statement (simple or compound) following the *test-condition* may be referred to as the Range of the **while-do** statement. Let us consider the case when the range consists of a compound statement as

```
    while test-condition do
          ┌begin
          │ ........
          │ ........
          │ ........
    range │ ........
          │ ........
          └end;
```

We represent this as

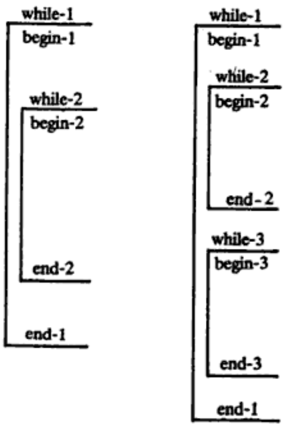


Similarly, we can represent the **for-do** statement with a compound statement as its range as:

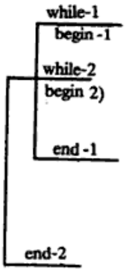


The following rules apply to the use of ranges of **while-do** and **for-do** in a program.

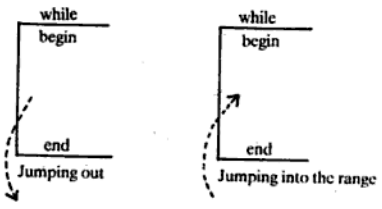
(i) the ranges may be nested as



(ii) the ranges should not cross one another as



(iii) Jumping out of the range is allowed but jumping in is forbidden. This is depicted below



Allowed Disallowed

If it is necessary that entry must be made in the range, then it should be made at the **while-do** or **for-do** statement.

We learnt about Fibonacci number sequence and their generation in Example 7.4. The program of this example was developed using control statements. Now we devise that program using **while-do** constructs.

Example 8.3

Develop a program to print the Fibonacci number sequence using **while-do** statement.

Program 8.3

```
program FIB (input, output);
  {Calculation of Fibonacci sequence using while-do statement}
    var
      K, I, J, L, N : integer ;
      { Main program begins }
    begin
      readln (N) { N is number of terms }
      K : =0 ; I : = 0 ;
      writeln (I) ;
      { Increment K and initialize J }
      K : = K+1 ; J : = 1 ;
      writeln (J) ;
      K : = K+1 ;
      { I, J, L give three successive Fibonacci numbers }
      { while loop begins, K is counter for number of terms }
      while K <= N do
        begin
          L : =I+J ; { calculation of next term }
          writeln (L) ;
          K : = K + 1 ; I : =J ; J : =L
        end
    end.
```

Sample input

```
6
```

Output

```
0
1
1
2
3
5
```

Rewrite the program of Example 8.1 using **while-do** construct.

8.3 The repeat-until Statement

This is another statement by which repetitive computations may be performed in a program. It also makes use of a test-condition like the **while-do**, but the test-condition is invoked at the end. With the availability of this statement, repetitive constructs in Pascal allow to program any type of situation without the use of **goto** statement. The format of the **repeat-until** statement is

```
repeat
  S1 ;─┐
  S2;  │
  .    │
  .    │ range
  .    │
  Sn ──┘
until test-condition;
```

where

$S_1, S_2, \ldots, S_n$ statements, may be simple or compound and form the range of the **repeat-until** statement.

The **repeat-until** statement operates as follows :
The statements S_1, S_2, ... S_n are executed first and then the test-condition is examined. If this value is false, statements S_1, S_2, ... S_n are executed again. This goes on till the value of the *test-condition* is true. Then control goes to the statement following the **until** keyword. Schematically, we can depict this action as shown in Fig. 8.5.

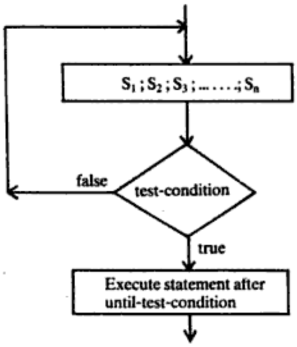

S1 ; S2 ; S3 ; ...... ; Sn
false
test-condition
true
Execute statement after until-test-condition

Fig. 8.5: Action of repeat-until statement.

The reader should draw the syntax chart of **repeat-until** statement himself.

An example of a **repeat-until** statement is

```
repeat
  SUM := SUM + X ;
  X := X - 2

until X < = 0;
```

Suppose, we wish to read 10 values of a number and write its squares. This can be easily done by the following loop :

```
count : = 1;
repeat
  readln (NUMBER);
  writeln (NUMBER * NUMBER);
  COUNT : = COUNT + 1

until COUNT > 10 ;
```

We illustrate further the use of **repeat-until** construct by an example wherein the method of iterations is used to compute the square root of a number.

Program 8.4

Develop an interactive program to compute the square root of a given positive real number by the method of iterations.

The method of iterations, also known as the Newton's method, to evaluate the square root of a number is as follows.

Let x be a number and y_1 be its approximate root. Then a better approximation for the root is given by the equation:

$$y_2 = (y_1 + x/y)/2.0$$

A still better value is obtained by substituting y_2 for each y_1 on the right hand side as :

$$y_3 = (y_2 + x/y_2)/2.0$$

And in general.

$$y_{n+1} = (y_n + x/y_n)/2.0$$

Calculation of y_n is continued till $y_{n+1} \simeq y_n$ upto to the desired accuracy. In our program, we impose the condition $[y_{n+1} - y_n] \leqslant 10^{-5}$.

The process is started by assuming some approximate value of y_1 to begin with. We shall take $y_1 = x/2.0$.

Program 8.4

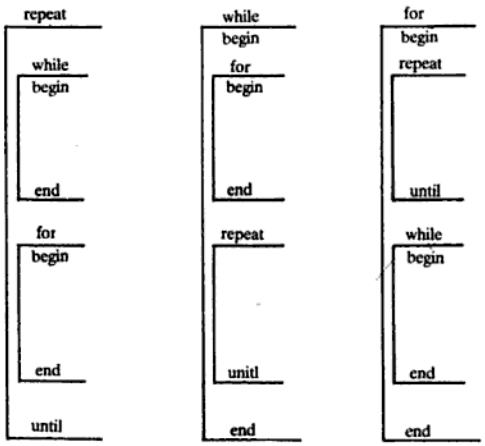
```
program ROOT (input, output);
  { Computation of square roots of real positive numbers by Newton's
  iteration method }

  var
    X, YN, YN1, DIFR : real ;
    CH : char ;

  begin
     repeat
        writeln ('Enter the number') ;
          readln (X) ;
          if X > 0.0 then
          begin
            YN : = 0.5*X ;
            repeat
              YN1 := 0.5 * (YN + X / YN) ;
              DIFR := YN1-YN ;
              YN : = YN1
            until (abs (DIFR) <= 1.0 E-05) ;
            writeln ('Squre root of', X : 3 : 1 '=', YN) ;
            writeln
          end ;
     write ('Do you want the root of another number? (Y/N') ) ;
     readln (CH) ;

   until (CHR = 'N') ;
   writeln ('Stop')

end.
```

Sample input/output

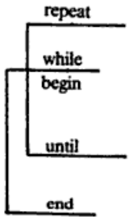
```
Enter the number
7.0
Square root of 7.0 = 2.64575 E + 00
Do you want the root of another number? (Y/N)
Y
Enter the number
13.0
Square root of 13.0 = 3.60555 E + 00
Do you want the root of another number? (Y/N)
N
Stop
```

The rules which applied to the ranges of **for-do** and **while-do** statements, also apply to the range of **repeat-until** statement. The range of this statement may consist of both simple and compound statements. Recall, that the ranges of the other two statements (**for-** and **while-**) can consist of either simple or compound statements, never both.

The ranges of the three repetitive statements may be nested among one another. Illustrations are



But crossing of ranges as



is generally disallowed.

When loops are nested, the innermost is executed fastest.

We observe that repetitive computations in a program may be performed, as governed by the values of logical expressions, using **while-do** and **repeat-until** constructs. The **while-do** and **repeat-until** statements are used quite extensively in developing programs. However, the following distinctions between these two statements should be kept in mind.

8.4 Differences between while-do and repeat-until statements

(i) The test-condition is examined, before the start of each loop execution, in the **while-do** statement, whereas in the **repeat-until** statement, the test-condition is tested after each execution of the range. This fact is emphasized in Pascal by specifying the test-condition with **while** at the start while the position of the test-condition is at the end with the **repeat** statement.

(ii) The **while** statement continues to be operative as long as the test-condition is true, but the **repeat** statement operates till the test-condition is false. Thus, the two statements examine the test-condition from opposite views.

(iii) The **repeat** loop is executed at least once, but the **while** loop may not be executed at all.

(iv) The **repeat** keyword is followed by a group of statements (one or more than one; each statement may be simple or compound) but the group appearing after the **while-do** always has only one statement which may be simple or compound.

The reader should note that the **for-do** loop is a deterministic loop, that is, the range is executed for a fixed number of times (decided before the execution starts), whereas **while-do** and **repeat-until** loops are indeterministic loops and their execution depends on certain test conditions.

The iterative constructs are very useful and are one of the building blocks for developing structured programs (Chapter 15). They, along with the arrays, are powerful tools to manipulate matrices, handle tables and large scale data processing applications.

Exercises 8

8.1. Complete the following sentences :

(a) The looping constructs available in Pascal are .... and ....
(b) The control variable in **for-do** statement cannot of .... type.
(c) In nested **for-do** loops, control variables must be ....
(d) The test-condition needs to be satisfied in the .... and .... iterative statements.
(e) The **while-do** statement is not executed even .... if the test-condition is .... at the start.
(f) The range of **repeat-until** statement can consist of both simple and .... statements.
(g) The execution of the **repeat-until** statement is complete only when the test-condition is found to be ....
(h) The ranges of two different iterative constructs can be ....

8.2. Answer the following as true or false :

(i) The value of the index variable in the **for-do** statement can either increase or decrease. (T/F)
(ii) The value of the test-condition in the **while-do** statement is computed only once at the start. (T/F)

(iii) The statement appearing in the **for-do** construct must be simple. (T/F)
(iv) When the **for-do** loop is completed, the value of the control variable is always unknown. (T/F)
(v) Transfer out of program execution from the range of repetitive statements is allowed. (T/F)
(vi) The use of enumerated type data is not allowed in the test-condition of statements **while-do** and **repeat-until.** (T/F)
(vii) Nesting of various iterative constructs is allowed. (T/F)
(viii) The crossing of ranges is generally not permitted. (T/F)

8.3. What are the various repetitive statements available in Pascal? Give and explain their formats and syntax diagrams.

8.4. Bring out the differences between **while-do** and **repeat-until** statements. Imagine and list the situations where one should be preferred over the other.

8.5. Can the action of a **while-do** statement be simulated by **if-then** or **if-then-else** statements? Illustrate by an example.

8.6. Explain the concept of range of a repetitive statement. Summarize the rules which govern the specification of ranges of such statements.

8.7. It is said that the control variable of the **for-do** statement should be of ordinal type. Bring out your reasons for and against this restriction.

8.8. Develop two program segments, using **while-do** and **repeat-until** statements, which read in 1000 data values or until a negative value is encountered, whichever comes first. Can you perform this action using **for-do** statements? How?

8.9. (a) Look at the following statements?

```
J := 0;
readln (K, N) ;
  repeat
    J := J+K ;
    M := N mod J ;
until M = 0 ;
```

Write a program segment using **while-do** construct to perform this action.

(b) Write the equivalent of

for J:= 1 **to** N **do** S_i :

using while-do construct.

8.10. Point out mistakes in the following statements. Write their correct forms.

```
(i) while no eof do
          S1; S2; ...... end ;

(ii) for J: = M down N do S1 ;

(iii) until X > Y .....
        repeat S1; S2; S3; ..... end

(iv) for A = 2.5 to 13.0 do S2 ;

(v) for J : = 1 to 10 do
       for J : = 5 downto 1 do
          write (J) ;
```

8.11. What outputs are obtained when the following program segments are executed (Here, J, K, M, N, P are assumed to be integer variables).

(a)
```
for J := 1 to 5
  for K := 1 do J
  writeln (J, K);
```

(b)
```
for M := 5 downto 1 do
    for N := 1 to M
      for P := 1 to N
        writeln (M, N, P, M * N * P);
```

(c)
```
J: = 1;
repeat
  M := J div 2 ;
  K := J * J ;
  J := J+1 ;
  writeln (J, M, K)
until J > 25 ;
```

8.12. Design a simple program to find the product of 1, 3, 9, 15, 21, 27, 33, 39.

8.13. Prepare a program that asks a question requiring the answer as Yes or No. Use the character Y for a yes and N for a no. Print the appropriate responses for the answers.

8.14. Design a program to print next year's calendar.

8.15. Develop a program to compute the root mean square of a set of given real numbers in a list. Use **eof** function for deciding the end of data values. Find also the number of data values in the list.

8.16. Read a positive number N and print a list of all those integers from 1 to N which are perfect squares.

8.17. Develop programs for Exercises 7.10, 7.17, 7.19 using any of the repetitive constructs.

8.18. The probability function of the Poisson distribution with parameter, λ, is given by

$$f(x) = \frac{\lambda^x}{x!} e^{-\lambda}$$

Develop a program to compute f(x) for x=0, 1, 2, 3, 4, 5, taking λ (lamda) = x/2.0. Print your results as

```
VALUE OF LAMDA =

X     F(X)
...   ...
...   ...
...   ...
```

8.19. Prepare a flowchart and a program to determine whether a given number contains any duplicate digits (For example, 12 has no duplicate digit, while 233 has 3 occuring as a duplicate).

8.20. Devise a Pascal program to draw a square and an equilateral triangle of arbitrary size.

8.21. Design a program to draw a straight line AB and then another line CD parallel to it.

8.22. Design programs to compute n! using the three looping constructs. Comment on the program as regards their length, efficiency and clarity.

8.23. Develop an algorithm and a program to compute the value of integral:

$$\int_1^3 \log_{10}(x)\,d\,x$$

using Trapezoidal rule, upto five decimals without using arrays. (see also Example 9.1).

8.24. Prepare a program to compute

$$\sum_{x=1}^{\infty} \frac{1}{x^3}$$

correct upto six decimals. Obtain the number of terms needed to attain this accuracy. Which iterative construct is most appropriate to design this program and why?

8.25. Design an interactive program for Example 7.2 using **repeat-until** construct only.

Chapter 9

Structured Data Type: Arrays

We have studied integer, real, boolean and character data type. These are the standard data types. The user-defined data types have been subrange and enumerated. All these are also referred to as Scalar Data types. Moreover, variables considered so far are called Simple or Unsubscripted variables. Such variables are not adequate to handle collections of data, tables, vectors/matrices, etc. Manipulation of tables and matrices is required in several applications, may be from statistics, engineering, science, mathematics, humanities, business, etc. This can be done with the help of subscripted variables and arrays. Subscripted variables may be used in programs in the same way as the simple variables.

An ordered collection of subscripted variables having the same name and other attributes defines an Array. An array is an example of Structured data type. Other varieties of structured data types available in Pascal are: Records, Files and Sets. Here, we shall consider arrays and subscripted variables and defer the discussion of other structured data types to later chapters.

9.1 Subscripted Variables

In mathematics, we are familiar with the notation: $x_1, x_2, x_3, \ldots.., x_{10}$. Here 1, 2, 3 . . . . ., 10 are the subscripts of variable x. We also say that variable x has 10 elements or components. These are different identifiers. In Pascal, we can represent them as x1, x2, x3 . . . x10, ... (or by some other name). Suppose x has 100 elements as: $x_1, x_2, \ldots.., x_{100}$. To use them in a Pascal program, we shall have to design 100 different variable names. This is not convenient. Pascal provides another way to represent such type of variables. This is done by placing the subscripts between two square brackets as: x[1], x[2], x[3], ..., x[100]. Such variables are referred to as Subscripted variables. Other examples of these variables are: SUM[10], B[25], TAB[1000] and so on. Each of these is a different variable. Contents of the square brackets are the subscripts and the characters appearing before the left square bracket constitute the name of the variable. Pascal allows any number of subcripts with a variable name. The general form may be specified as:

name [$s_1, s_2, s_3, \ldots..$]

where

name	is the name of the subscripted variable,
$s_1, s_2, s_3, \ldots..$	are the subscripts,

name of a subscripted variable is formed according to the rules of constructing variable identifiers. The subscripts s_1, s_2, s_3, ... must be separated from each other by commas, enclosed within square brackets. Value of the subscript may be negative, positive or zero.

The subscripts of a variable may be

- integer constants
- simple or subscripted integer variables or integer expressions
- variables of subrange or enumerated type.

These points are illustrated by the syntax chart of Fig. 9.1.

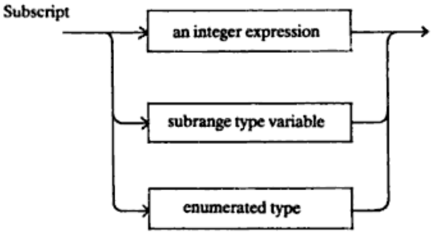


Fig. 9.1: Syntax chart of subscript

9.2 Arrays

In the beginning of this chapter, we had given the definition of an array as: an ordered collection of subscripted variables having the same name and other attributes such as real, integer, character, boolean, etc. Let us understand it further by a simple example.

Consider the set of subscripted variables: A[1], A[2], A[3], A[4], A[5]. The name A is common to all. A is called an array. A[1], A[2], ..., A[5] are its elements. A[1] is its first element and A[5] is the last element. 1, 2, 3, 4, 5 are the subscripts. When the elements of an array have single subscript, the array is called a Linear or one dimensional array or a List. Here A is a linear array. A vector is another example of a linear array.

In a two dimensional array, each element has two subscripts; for a three dimensional array, there are three subscripts with each element, and so on. Pascal does not put any limit on the number of subscripts, though their number may be implementation dependent.

We have seen that the type of every variable used in a Pascal program must be declared. This type declaration has to be specified for an array as well. Moreover, the Pascal compiler has to be supplied information about the number of elements in an array (or the size of an array) and their type. This can be done by defining array type and array variables.

The general form of type declaration of one-dimensional array is

---

type
array-type-identifier = **array** [t_1] **of** t_2 (I)

---

where

array-type-identifier	indicates the type name
t_1	specifies the data type of the values to be used as subscripts; it must be a scalar data type except real
t_2	indicates the type of values that are going to be stored in the array elements, that is, it specifies the type of array elements; it may be of any type real, boolean, integer, char, subrange or enumerated.

An example of array type declaration is

type
TABLE = **array** [1 .. 25] **of integer** (II)

Here the subscript is of subrange type and elements of array are of type integer. The identifier TABLE can be used to define array variables. This is illustrated in the following.

Once we have defined an array data type, we can create actual array variables, that is arrays, using the declaration **var** as

---

var
v_1, v_2, ...: *array-type-identifier*

---

where v_1, v_2, ... are the variable names and *array-type-identifier* is the identifier which has been defined under the **type** declarations for an array, as in (I).

Suppose we wish to declare A as an array of 25 elements of integers. This can be done as

var
A : TABLE

In other words, A is an array variable of type TABLE defined in (II) above.

The first element of A is accessed as A[1], second element as A[2], third as A[3], ...and 25th element as A[25]. If you specify A[26] or A[0], there will be an error because the subscript can lie only in the range 1 to 25.

Now consider the following example :

```
type
  CURRENT = array [-10..10] of real;
var
  I, J : CURRENT;
```

Here I and J are array variables of type CURRENT. Their elements are I[−10], I[−9], I[−8], . . . I[0], I[9], ... ,I[10], that is, 21 elements. Similarly, for J. Values assumed by the elements of arrays I and J must be of type real.

The type of subscripts can also be character, boolean, enumerated or subrange. Look at the example

```
type
  CHRX = array [char] of 1..64;
var
  X : CHRX;
```

Here X is an array of type CHRX whose elements can be accessed as: X['A'], X['B'], X['C'], . . . , X['Z'], X['0'], X['1'], . . . X['9'] if **char** indicates the characters 'A', 'B', . . ., 'Z', '0' '1', . . ., '9'.

However, values which the elements of array X can assume must lie in the range 1 .. 64. We can specify, for instance,

```
for M : = 'A' to 'Z' do
  X [M] : = ord [M] ;
```

where M has been assumed to be a variable of type **char**.

In the example

```
type
  FF = array [FLOWER] of boolean;
var
  RED, WHITE : FF;
```

RED and WHITE are array variables of type FF. Suppose FLOWER is of enumerated type, that is,

```
FLOWER = (ROSE, LILY, NARGIS)
```

then the elements of array RED and WHITE will the RED [ROSE], RED [LILY], RED [NARGIS] and WHITE [ROSE], WHITE [LILY], WHITE [NARGIS] respectively. The values that these elements may be assigned must either be true or false. Let us specify

```
RED [ROSE] := true ; RED [LILY] := false ;
```

Consider the following statement

```
if RED [ROSE] and RED [LILY] then
  WHITE [NARGIS] := RED [ROSE]
else
  WHITE [ROSE] := RED[LILY];
```

Here, element WHITE [ROSE] is initialized to **false** as the test-condition RED [ROSE] **and** RED [LILY] is false.

Arrays (or array variables) may be defined directly as

```
var
 v1, v2, .....: array [t1] of t2                                (III)
```

where v_1 and v_2 are the array variables; t_1 and t_2 have the same significance as defined earlier with **type** declaration. Examples are

```
var
  VOLTAGE : array [1 .. 10] of real;
  VECTOR : array [1 .. 100] of −10 .. 10;
  PAGE : array [1 .. 25] of char;
```

In the declaration (I) and (III), t_1 and t_2, either or both, can be either data type names or data type definitions.
To understand this, look at the following example

```
type
  COLOUR = (RED, BLUE, WHITE, GREEN) ;
  P = array  [COLOUR]  of integer ;                          (IV)
```

Instead of declaring type P as above, it can also be specified as:

```
type
  P = array [ (RED, BLUE, WHITE, GREEN)] of integer;   (V)
```

In definition (IV), data type name COLOUR has been used, while in (V), data type definitions have been specified. This type of declarations are also accepted by Pascal. The array variables can be declared, [as of type P, in (IV)] in the usual way under the declaration **var**.

Arrays and their sizes may also be declared as

```
const N = 100 ;
type INDEX = 1 .. N ;
var A : array [INDEX]  of integer ;
```

Now array A has 100 elements as N = 100. By changing N, elements of array can be varied.

We further illustrate the declaration and use of arrays by a simple example.

Suppose there are 10 data values of type real. We want to store them under a common name and compute the sum of these. The program segment for this appears as:

```
program SUM (input, output) ;
  { Declare identifier X of array type }

    type
      X = array [1 .. 10] of real;
```

```
var
   LIST : X ; { LIST is an array variable having 10 elements }
   M : integer ; S : real ;
begin
   S := 0.0
   for M := 1 to 10 do
      begin
         readln (LIST [M] ) ;
         S := S + LIST [M] { Value of sum is stored in S }
      end;
      writeln ('Total of 10 data values = ', S)
end.
```

Next we demonstrate the use of arrays by an example of numerical integration. The simplest way to carry out integration of any smoothly varying function is the trapezoidal method. This method is explained below and its program developed using arrays.

Example 9.1.

Develop a program to evaluate the following integral

$$\int_0^{4.0} x(x+1)\,dx$$

by the trapezoidal rule. Divide the interval [0 – 4.0] in 100 equal steps.

According to the trapezoidal rule, the integral $\int_a^b f(x)dx$ is approximated as:

$$I = \int_a^b f(x)\,dx = \tfrac{1}{2} \sum_{i=1}^{n} h_i \left(y_i + y_{i+1}\right) \qquad (1)$$

where y_i are the values of the function f(x) at points x_i. Here $h_i = x_{i+1} - x_i$ is the spacing between points x_{i+1} and x_i ; a, b are the lower and upper limits respectively. Equation (1) is general. When the spacing is equal, it becomes

$$I = \frac{h}{2}\left[\left(y_1+y_2\right) + \left(y_2+y_3\right) + \ldots. + \left(y_{n-1}+y_n\right) + y_{n+1}\right] \qquad (2)$$

$$= h\left[\left(y_1+y_{n+1}\right) \big/ 2.0 + \left(y_2+y_3 + \ldots.. + y_n\right) \right] \qquad (3)$$

where, for equally spaced values of x, y_i are the values of f(x) at a, a + h, a + 2h . . ., b = a + nh. Now h = (b–a)/n.

The program for Example 9.1 may appear as follows :

Program 9.1

```
program TRAPEZ (input, output) ;
   { To integrate X * (X + 1) by trapezoidal rule between the limits 0 – 4.0 }
   { Store integrand in array Y. Say, A = lower limit, B = upper limit, N =
   number of steps }
   const
      MAX = 400;

   var
      Y: array [1 .. MAX] of real;
      J, M, N : integer;
      A, B, SUM, X, VAL, H, Y1YM : real ;
   begin
      writeln ('Enter values of A, B, N') ;
      readln (A, B, N);
      H : = (B–A)/N ; { step size }
      M : = N + 1 ;
      X : = A ;
      J : = 1 ;
   repeat
      Y [J]: = X * (X + 1);
      X: = X + H;
      J: = J + 1
   until J > M;

      Y1YM : = (Y [1] + Y [M] ) * 0.5 ;
      SUM : = 0.0 ;
      for J : = 2 to N do
         SUM : = SUM + Y[J] ;
      VAL : = (SUM + Y1YM) * H ;
      writeln ('Lower limit =', A : 6 : 2) ;
      writeln ('Upper limit =', B : 6 : 2 ) ;
      writeln ('Step size =', H : 6 : 2) ;
      writeln ('The integral is =', VAL : 8 : 4)
   end.
```

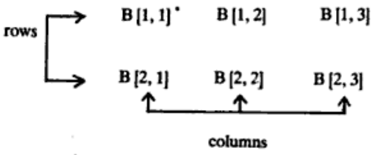
Sample input/output

```
Enter values of A, B, N
0.0   4.0   100
Lower limit = ƀƀ0.00
Upper limit = ƀƀ4.00
Step size = ƀƀ0.04
The integral is = ƀ29.3344
```

Run this program for N = 200, 300 and 400. Compare the values of integrals. Which is the most accurate?

9.3 Two-dimensional Arrays

Two-dimensional arrays have two subscrips. The first subscript represents rows and the second subscript refers to the columns (this is by convention). Thus, if B is a two-dimensional array having two rows and three columns, then its elements may be indicated as



Due to this representation, a two-dimensional array is also called a Table or a Rectangular array. Matrices are very good examples of two-dimensional arrays.

Two-dimensional arrays may be declared as

```
type
  array-type-identifier = array [$t_1$, $t_2$] of $t_3$
var
  $v_1$ $v_2$ . . . : array-type-identifier
```

or as

```
var
  $v_1$, $v_2$ . . . . : array [$t_1$, $t_2$]  of  $t_3$
```

where *array-type-identifier*, t_1, t_2, t_3 and variables v_1, v_2, . . have the same meaning as before, except that now, one more type, indicated by t_2, has been added. t_3 is also called Base type and specifies the type of elements of array variable. The first subscript of the array should be of type t_1 while the second is to be of type t_2. Types t_1, t_2, t_3 may be identical or different.

Suppose, we wish to define the above mentioned array B as a two-dimensional array, whose elements can store real data. This may e done as :

```
type
  XTWO = array [1 . . 2, 1 . . 3]  of real;

var
  B: XTWO;
```

or directly as

```
var
  B: array [1 . . 2, 1 . . 3]  of real;
```

or as

```
const M = 2 ; N = 3 ;
type S1 = 1 .. M ;
     S2 = 1 .. N ;
var B : array [S1, S2] of real ;
```

Now look at the declarations

```
type
  SUBJECTS = (HINDI, PUNJABJI, BENGALI, TELUGU, TAMIL) ;
  TAB = array [1 .. 25, SUBJECTS] of char ;
var
  X, Y : TAB;
  LANGUAGE : SUBJECTS ;
```

Here, X and Y are array variables of type TAB (two-dimensional), while LANGUAGE is a simple enumerated type variable. The first subscript of arrays X and Y can vary from 1 to 25, while the second can assume the values HINDI, PUNJABI, BENGALI, TELUGU and TAMIL. The reader should note here that the two subscripts are of different type. In fact, the subscripts of an array variable need not be of the same type. This facility makes Pascal programs better documented as the subscripts—type and name-can be selected according to the application. This will further become clear as we proceed. Other examples of two-dimensional array declarations are :

```
var
  STUDENT : array [1 .. 1000, SUBJECTS] of integer;
  MATRIX : array [1 .. 25, 1 .. 10] of real;
  DAYSINYEAR : array [MONTHS, 1 .. N] of 28 .. 31;
  APAGE : array [LINE, WORDS] of char ;
```

Thus, the reader should observe that there is no restriction at all on the subscript type of an array, however, all the elements of an array must be assigned data according to the element type. So, while using arrays, care must be taken about

(i) the values which the subscript or index can assume,
(ii) the values which the array elements can take.

We explain this point by an example of a one-dimensional array for reasons of simplicity and clarity.

```
type
  MONTH = (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG,
           SEP, OCT, NOV, DEC) ;

var
  DAYS : array [MONTH] of 28 .. 31;
  MONTHNAME : MONTH;
```

Here DAYS is a one-dimensional array. Its elements are DAYS [JAN], DAYS [FEB], DAYS [MAR], ... DAYS [DEC]. Values which these elements can assume must lie between 28 to 31 (inclusive of 28 and 31). We may specify

```
    DAYS [JAN] := 31
    DAYS [FEB] := 28
    DAYS [MAR] := 31
      . .
      . .
      . .
    DAYS [DEC] = 31
```

Suppose you wish to find the total number of days in a non-leap year. A simple statement for this may be as:

```
    TOTALDAYS : = 0;
    for MONTHNAME = JAN to DEC do
        TOTALDAYS : = TOTALDAYS + DAYS [MONTHNAME];
```

Example 9.2

Given two matrices MATX and MATY with elements defined as:

MATX (J, K) = J + K   J = 1, 2, ..., 5
MATY (J, K) = J * K   K = 1, 2, ..., 5

The elements of product matrix MATZ are defined as

$$\text{MATZ}(J,K) = \sum_{L} \text{MATX}(J,L) * \text{MATY}(L,K)$$

Develop a program to compute MATZ. Print the values of elements of matrices MATX, MATY, MATZ in a tabular form with appropriate headings.

A simple program to obtain the product of the given matrices is as follows:

Program 9.2

```
program MATMUL (input, output) ;
  var

    MATX, MATY, MATZ : array [1 .. 10, 1 .. 10] of integer;
    I, J, K, L, M, N, SUM : integer;

  begin
  { Compute the elements of matrices MATX and MATY }
  for J : = 1 to 5 do
    for K : 1 to 5 do
        begin
          MATX [J, K] : = J + K ;
          MATY [J, K] : = J * K
        end;
```

```
{ Computation of product of matrices MATX and MATY }
  for J : = 1 to 5 do
     for K : = 1 to 5 do
         begin
           SUM : = 0;
           for L : = 1 to 5 do
               SUM : = SUM + MATX [J, L] * MATY [L, K] ;
               MATZ [J, K] : = SUM
         end ; { MATZ contains the product }
{ the values of matrices are written below }
  writeln;
  writeln ('      MATX') ;
  writeln ('      - - - -')
  writeln;
  for J : = 1 to 5 do

      begin
        for K : = 1 to 5 do
           write (MATX [J, K] : 5);
        writeln
      end;
  wirteln; writeln ('      MATY') ;
  writeln ('      - - - -');   writeln;
  for J : = 1 to 5 do
     begin
       for K : = 1 to 5 do
          write (MATY [J, K] : 5);
  writeln
end;
  writeln; writeln; ('      MATZ );
  writeln  ('      - - - -');
  writeln;
  for J : = 1 to 5 do
     begin
       for K : = 1 to 5 do
          write  (MATZ [J, K]  : 5) ;
          writeln
end
end.
```

Sample output

```
     MATX
     - - - -
     2    3    4    5    6
     3    4    5    6    7
     4    5    6    7    8
     5    6    7    8    9
     6    7    8    9   10
```

```
MATY
----
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25

 MATZ
 ----
  70 140 210 280 350
  85 170 255 340 425
 100 200 300 400 500
 115 230 345 460 575
 130 260 390 520 650
```

The example Program 9.2 illustrates the generation, multiplication and output of matrices (tables) in a simple way. We have considered only 5 × 5 square matrices in the example. The program can be written in such a way that product is formed for any size of two matrices, provided the number of columns of first matrix is equal to the number of rows of the second matrix. The student is urged to modify Program 9.2 to compute the product of two matrices A and B of arbitrary size. The Program should have built-in check to examine whether the above requirement is satisfied and only then product is calculated.

9.4 Multidimensional Arrays

In many engineering, scientific and other applications, we need to deal with three-, four, or higher dimensional arrays. Such arrays allow greater and more compact representation of interrelationships between data objects. Pascal allows arrays upto any dimensions, however, restriction may be imposed during its implementation on a particular computer system. Multidimensional arrays may be declared in a program in a similar way as we discussed the declaration of 1- and 2- dimensional arrays above. The general format is

type
array-type-identifier = **array** [t_1, t_2, t_3, ...] **of** t ;
var
v_1, v_2, ... *array-type-identifier;*

Here t_1, t_2, t_3, ... indicate the type of subscripts. They may be identical or different. t specifies the base type. It may be same as t_1, t_2, t_3, ... or different.

An example of 3-dimensional declaration is

```
type
  ROOM = array [LENGTH, BREADTH, HEIGHT] of 10 .. 15;
var
  ROOMX, ROOMY, ROOMZ : ROOM ;
```

Another illustration is

```
const
  PAGES = 500 ; LINES = 40 ; WORDS = 13 ;
var
  BOOK : array [1 .. PAGES, 1 . . LINES, 1 .. WORDS] of integer ;
```

If we specify

```
BOOK [13, 7, 9]
```

it specifies 13th page, 7th line on that page and 9th word in that line.

Further example is

```
type
  GALAXY = array [1 .. 100, (MOON, SUN, EARTH, MARS,
    JUPITER, NEPTUNE), (HOT, COLD, WARM) ] of integer;
var
  TIME, DISTANCE : GALAXY ;
```

Here TIME and DISTANCE are arrays of type GALAXY. Each has three subscripts and hence are three-dimensional arrays.

Examples of indicating some elements are :

```
DISTANCE [1, MOON, COLD]
..........
..........
..........
DISTANCE [5, EARTH, WARM]
..........
..........
..........
TIME [4, SUN, HOT]
..........
..........
..........
```

Higher dimensional arrays can be defined in the same way as explained above.

The elements of an array can be simple or arrays themselves. Thus, it is valid to define

```
type
   TABLE = array [1 .. M]  of array [1 .. N]  of integer;
```

This is equivalent to

```
TABLE = array [1 .. M, 1 .. N] of integer;
```

Thus, TABLE is a two-dimensional array. An alternative specification of a two-dimensional array may also be given as

type
array-type-identifier = **array** [t_1] **of array** [t_2] **of** t_3

and, in general, we can specify a multi-dimensional array type as

type
array-type-identifier = **array** [t_1] **of array** [t_2] **of array** [t_3] . . . . **of** t

This is the same as given earlier as

type
array-type-identifier = **array** [$t_1, t_2, t_3, \ldots .$] **of** t

9.5 Compatible Arrays

Arrays are compatible when they are of identical type. Such arrays may be declared with the same type declaration or with the same variable declaration. Consider the example

```
type
   DATAINPUT = array [1 .. 80] of char;
DATAOUTPUT = array [1 .. 132] of char;

var
   INPUTLINE1, INPUTLINE2 : DATAINPUT ;
   OUTPUTLINE1, OUTPULINE2 : DATAOUTPUT ;
```

Here array variables INPUTLINE1 and INPUTLINE2 are of identical **type** and hence are compatible. Similarly, arrays OUTPUTLINE1 and OUTPUTLINE2 are compatible. For compatible arrays, an assignment of the type

```
INPUTLINE2 := INPUTLINE1
```

is allowed. This statement assigns the values of each element of array INPUTLINE1 to the corresponding element of array INPUTLINE2. In fact, this is equivalent to

```
for K := 1 to 80 do
   INPUTLINE2 [K] : = INPUTLINE1 [K];
```

Compatible arrays are useful to transfer data from one array to another array. This makes program compact and more efficient.

We illustrate the use of compatible arrays (though simple) and bring out the necessity of array data structure for solving certain kind of problems. This concerns sorting of data. We explain the procedure first and then develop a program. For explanation purposes, we have taken the data to be numeric, but the algorithm is applicable to non-numeric data as well.

Consider the following list of numbers :

6, 5, 8, 4

When these are arranged as

8, 6, 5, 4

they are in descending order. If we write them as

4, 5, 6, 8

they are in ascending sequence.

Ordering of data as given above is referred to as Sorting. The problem of sorting is encountered in several applications, such as finding the median of statistical data, arranging words alphabetically, etc. There are many methods to sort data, but the simplest of all is the method of interchanges. The various steps involved in this procedure to arrange numbers in ascending sequence are :

1. Scan the list of numbers from left to right·
2. Compare every number with its right-hand neighbour.
3. If the number on the right is smaller than the number on the left, interchange them, otherwise leave them as they are.
4. Carryout steps 1-3 till all the numbers have been arranged.

The above procedure is explained further by an example. Look at the numbers

6, 5, 8,4

Scan 1

(a) Compare 6 and 5. As 6 is greater than 5; interchange them. The list is now

5, 6, 8, 4

(b) Now compare 6 and 8. As 6 is less than 8, do not interchange them. The list remains

5, 6, 8, 4

(c) Next compare 8 and 4. As 8 is greater than 4, interchange them. The list becomes

5, 6, 4, 8

Thus, the largest number has moved to the extreme right position in one scan. Note that, as there are four numbers in the list, three comparisons are required as indicated by steps (a), (b), (c).

Scan 2

Now repeat the above process with the new list

5, 6, 4, 8

assuming that 8 has been 'deleted'. It requires only two comparisons, as we need to arrange 5, 6, 4 only. At its conclusion, the list appear as

5, 4, 6, 8

Scan 3

Next, perform the same process as in *Scan 2*, assuming 6 and 8 have been 'deleted', that is, they need not be compared as they are already in the ascending order. Now comparison is to be made between 5 and 4 only. There is only one comparison. The final list becomes.

4, 5, 6, 8

Thus, the four numbers are arranged in the ascending order in three scans. In general, if there n numbers in the list, then (n−1) scans are required to place them in the ascending/descending sequence.

We see that the numbers to be sorted must be available together in a list. This is done by reading all the numbers in an array. Here is an example of a problem where use of arrays is must.

Implementation of exchange algorithm is described below.

Example 9.3

Given a list of numbers (say, 7)

6, 4, 5, 12, 1, 13, −5

Develop a general program to do the following :

(i) store the unsorted numbers in array A.
(ii) sort the numbers in ascending sequence and store in array B.
(iii) compute $\sum_{i=1}^{N} (|A_i - B_i|)$

Let us define two arrays A and B, each having (say) 500 elements. Data are read into array A. Next, data are transferred into array B and then B is used for sorting routine.

Program 9.3

```
program EXCHSORT (input, output) ;
{ Sorting of data by exchange alogrithm }
var
  I, J, K, N, TEMP, SUM : integer ;
               A, B : array [1 .. 500] of integer ;
begin
  write ('Number of elements in the list =') ;
  readln (N) ; writeln ;
  writeln ('The unsorted list is') ; writeln ;
for J : = 1 to N do
  read (A [J]) ; { array A contains the original data }
B := A : { array B is initialized to array A ; A and B are compatible arrays }
{ Data are sorted below and stored in array B }
for J := 1 to N do
  begin
    for K : = 1 to N−1 do
      begin
        if B [K] > B [K + 1] then
          begin { elements are interchanged below }
            TEMP := B [K + 1] ;
            B [K] := B [K + 1] ;
            B [K + 1] := TEMP
          end
        end ; { sorting is complete }
        writeln
      end ;
writeln ('The sorted list is') ; writeln ;
for J := 1 to N do write (B[J]), ('b')) ;
{ computation of sum of absolute differences of corresponding elements }
SUM : = 0 ;
for J := 1 to N do
    SUM := SUM+abs (A[J]−B[J]) ;
writeln ('Sum of absolute differences =', SUM)
end.
```

Sample input/output

```
Number of elements in the list = 7
The unsorted list is

6   4   5   12   1   13   −5

The sorted list is

−5   1   4   5   6   12   13

Sum of absolute differences = 46
```

9.6 Packed Arrays

Normally, characters, truth values (false and true) etc. are stored in such a way that each item occupies one storage unit (normally a word). This is referred to as Unpacked mode of data storage. As an illustration, let us see how the word BEAUTY is stored. Assuming each unit (word) contains 4 bytes and each byte represents one character, then storage of BEAUTY appears as shown in Fig. 9.2(a) One byte of each memory word is used while three other bytes remain unused. Thus, this mode of data storage, that is, unpacked mode, is uneconomical and considerable amount of storage goes as unutilized. Storage is fully utilized if characters are stored in consecutive bytes as shown in Fig. 9.2(b). This mode of data storage is referred to as Packed mode.

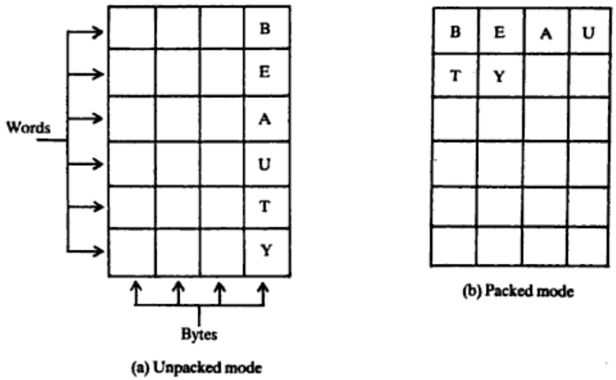


Fig. 9.2: Data storage mode (each square represents a byte)

Pascal allows the arrav data to be stored in packed mode. The declaration to do so has the form:

type
array-type-identifier = **packed array** [t_1, t_2, ...] **of** t;

var
v_1, v_2, ... : *array-type-identifier*;

or as

var
v_1, v_2, ... : **packed array** [t_1, t_2 , ...] **of** t;

where *array-type-identifier*, t_1, t_2 . . . ., t and v_1, v_2, ... have the same meaning as described in Section 9.4. Now the attribute **packed** has been added before the keyword **array**. The attribute **packed** may be used with any type of data; **integer, real, boolean, char,** or user-defined.

An example of packed array declaration is :

```
type
  PP = packed array [1 .. M]  of boolean ;
var
  X, Y : PP ;
```

or alternatively as

```
var
  X, Y : packed array [1 .. M]  of boolean;
```

Variables X and Y are arrays which can store boolean data values for the elements in a packed mode. Similarly, if we declare

```
var
  MAT1, MAT2 : packed array [1 .. M, 1 .. N]  of integer ;
```

then values (integer) of elements of arrays MAT1 and MAT2 are stored in packed form.

Attribute **packed** may be used with an array of any dimension.

User is not concerned with how the values are packed and stored. It is the responsibility of the Pascal compiler. It will automatically pack the values and store them in memory. Individual elements of packed arrays can be accessed in the same way as of unpacked arrays, except that the access time is increased for packed array elements. However, advantage of using packed arrays is that storage space is saved. Generally, the attribute **packed** is used when memory size constraints are there, which normally arise for large dimensioned variables. Moreover, packed arrays of type **char** and **boolean** find applications in many areas, such as word processing, stock inventories, mailing list, etc.

Data stored in a packed array may be assigned to an unpacked array. For example, consider

```
var
  X : packed array [1 .. 100]  of integer;
  Y : array [1 .. 100]  of integer;
```

If we write

```
for J := 1 to 100 do
  Y [J] := X [J];
```

then values of packed array X are stored in array Y in an unpacked form.

Some Pascal implementations may provide a single instruction that performs the conversion operations, that is, packed to unpacked and unpacked to packed offer the facility of defining string data type, associated operators and functions. Reference to the computer centre manual may be made to ascertain the correct format of declaring the string data type.

Packed array offer an alternative facility to define strings. A linear packed array of characters is defined as a String. The number of characters that can be stored in such an array is equal to the number of elements of the array.

An example of a string is

```
var
  NAME : packed array [1 .. 15] of char;
```

Here NAME is a string which can consist of 15 characters. Further examples are

```
var
  TEXTBOOK1, TEXTBOOK2 : packed array [1 .. 10] of char ;
```

Here TEXTBOOK1 and TEXTBOOK2 are the string variables, each of length 10 characters. We may make assignments to these via string constants. An example is

```
TEXTBOOK1 := 'PHYSICS' ;
  TEXTBOOK2 := 'MATHEMATICS' ;
```

In the case of string constant 'PHYSICS', length is 7, so when PHYSICS is stored as the value of variable TEXTBOOK1, the system adds three blanks on the right as indicated in Fig.9.3(a).

This is referred to as Padding. It is always done on the right side. The various elements of array TEXTBOOK1 have the values as shown in Fig.9.3(b).

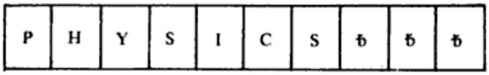


Fig. 9.3(a)

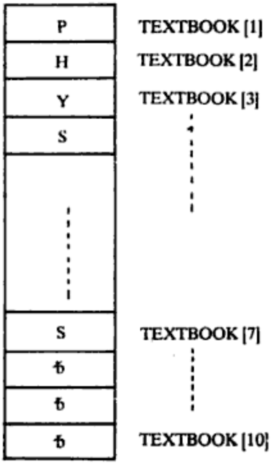


Fig. 9.3(b)

The length of the constant 'MATHEMATICS' is 11, so when this is stored as the value of the variable, TEXTBOOK2, the stored data appear as:

M	A	T	H	E	M	A	T	I	C

The rightmost character S is truncated. The reader should list the values of various elements of array TEXTBOOK2.

9.8 Operations on Strings

Many operations may be performed on strings, such as comparison, concatenation (joining of strings), splitting of strings, finding patterns in a string, presence of a substring in another string, and so on. This is also referred to as String processing. Strings may be processed by storing character data in linear packed arrays. Any element of array can be accessed, altered or deleted.

Strings may be compared using the relational operators in the same way as we compare individual characters, however, the strings to be compared should be of the same length. Strings are treated as units for such operations. During comparison of strings, the Pascal compiler automatically checks the alphabetical ordering. If the strings are not identical, then the first pair of characters in the corresponding positions of the two arrays, which differ, is the pair that decides the relationship. This depends on the collating sequence of charactrers available in a computer.

For the ASCII character set (see Appendix VI),

'RAMLAL' > 'RAMJEE'

because 'L' follows 'J'. Similarly,

'456' < '576'

is true (?). However, 'RAM' < 'SHAM' is not allowed as the strings are of different length. Thus, remember, comparison of only packed character arrays, which are compatible, is allowed. (Some Pascal version may not insist on this requirement.)

Any operation allowed on an array variable is also permitted on a string because it is a packed character array. Moreover, *it should be noted that the name of the packed character array variable is the name of the string.*

Strings cannot be read with **read** statement, but they can be written with **write** statement on the output device. For example, we cannot specify

read (TEXTBOOK1)

but the statement

write (TEXTBOOK1)

is allowed. However, if an array of type character is to be read, then it is necessary to do so character by character, testing all the time whether the end of line has been reached. These points are illustrated by the program of Example 9.4.

Pascal does not define a concatenation operator to concatenate strings. A concatenated string can be obtained from shorter strings by copying the smaller strings, character by character, into a larger string, i.e. packed character array. This is illustrated by the example given below.

```
var
   STRING : packed array [1 .. 10]  of char;
   STR1, STR2  : packed array [1 .. 5]  of char;
      J : integer;

      ..........
      ..........
      ..........

      for J := 1 to 5 do
            STRING [J] := STR1 [J] ;
      for J := 6 to 10 do
            STRING [J] := STR2 [J – 5] ;
```

These statements transfer the values of elements of array STR1 into the first 5 elements of composite string STRING. Next, the values of elements of array STR2 are transferred into the elements 6, 7, 8, 9, 10th of array STRING.

The above technique can also be used to select a substring of a bigger string. A substring is defined as the consecutive characters in a string. For instance, in the string,

'ALWAYS SPEAK THE TRUTH'

ALWAYS, WAYS, SPEAK, THE, TRUTH or RUTH, and so on, can be substrings as they consist of consecutive characters. But combinations of isolated characters, such as WASTE, are not substrings of the above string.

The method to select a substring of a string is to copy the required portion of a string, again character by character, into another string. This is illustrated by the following example:

```
var
   STRING : packed array [1 .. 10] of char;
   SUBSTRING : packed array [1 .. 4] of char;
```

The characters 3-6 of array STRING may be transferred into array SUBSTRING by the following statement

```
for J := 3 to 6 do

   SUBSTRING [J-2] := STRING [J] ;
```

Next, we present an example which illustrates the search and count of a substring in a given text.

Example 9.4

Develop a program which reads a paragraph (terminated by an asterisk, *, symbol) and then searches the occurrence and frequency of a given substring in the paragraph.

Program 9.4

```
program SUBSTR (input, output) ;
   { Program to count the number of times a substring occurs in a string of
character }

const
   MAXSUBSTR = 20; { gives maximum length of the substring }
MAXPARA   = 1000; { gives maximum length of the paragraph }

var
   I, J, M, N, COUNT : integer;
   SUBSTRING : packed array [1 .. MAXSUBSTR] of char ;
   PARA : packed array [1 .. MAXPARA] of char ;
   FLAG : boolean ; OPTION : char ;

begin
   writeln;
   writeln; ('Enter the paragraph. Terminate it with *');
   M : = 1 ;
   read (PARA [M] );
   while (PARA [M]  < >  '*')  do

      begin
         M : = M + 1;
         read (PARA [M])
      end; { The paragraph has been read in array PA RA }

repeat
   writeln ('Enter the substring. Terminate it with *') ;
      N : = 1;
   read (SUBSTRING [N]);
      while (SUBSTRING [N] < > '*') do
         begin
            N : = N + 1;
            read (SUBSTRING [N])
         end;

{ The substring is searched below }

   N : = N - 1;
   I :=0 ; COUNT : = 0 ;
   while ( I < = M) do
```

```
          begin
            I:=I+1;
            FLAG:=true;
            J:=1
            while (FLAG and J <=N)  do

              begin
                if (PARA [I] = SUBSTRING [J]) then

                  begin
                    I:=I+1;
                    J:=J+1;
                    if (J > N) then
                      if (PARA [I] < > ' ') and
                        (PARA [I] < > '.') and
                        (PARA [I] < > ',') then
                          FLAG:=false
                  end;
                                 else FLAG: false
            end;

        if (FLAG = true)  then COUNT:= COUNT + 1
          else while (PARA [I] < > ' ') and
            (PARA [I] < > ',') and
            (PARA [I] < > ('.') do
              I:=I+1
      end;

        write ('The number of times') ;
        for I:= 1  to  N  do
          write (SUBSTRING [I]) ;
        write ('occurs', COUNT, '.') ;
        writeln;
        writeln ('Do you wish to give another substring? [Y/N]');
        readln (OPTION)
        until (OPTION = 'N')
        writeln ('Stop')
  end.
```

Sample input/output

```
Enter the paragraph. Terminate it with *
HOW ARE YOU? FINE. WE ARE MEETING AFTER A VERY LONG
TIME. TRUST, YOU AND OTHER MEMBERS OF YOUR FAMILY
HAVE BEEN FINE. *
Enter the substring. Terminate it with *
FINE *
```

(d) All elements of an array need not be of identical type. (T/F).
(e) Any element of an array must be accessed sequentially. (T/F)
(f) The type of array identifier and subscript identifiers must be same. (T/F)
(g) Compatible arrays need not be of identical type. (T/F)
(h) Packed arrays help to save computer memory. (T/F)
(i) Packed array data cannot be assigned to unpacked arrays. (T/F)
(j) Strings are handled in Pascal via packed character arrays. (T/F)
(k) Data type name and data type definitions are same. (T/F)

9.2. Complete the following sentences.

(i) The value of subscripts of a variable can be ......,.......,.....
(ii) A linear array is also called ....
(iii) A matrix is a .... array.
(iv) An ordered collection of subscripted variables having the same .... and other .... is called an ....
(v) The indices of subscripted variables are always enclosed between .... brackets.
(vi) Attribute **packed** is used when .... constraints are there.
(vii) A character variable can store only one ....
(viii) The Pascal compiler automatically examines .... during comparison of strings.
(ix) In the unpacked mode of character storage, each character occupies .... of computer memory.
(x) The packed data can be changed into the unpacked form using the .... statement.

9.3. Explain the difference between subscripted variables and arrays. Give examples of 1- 2-, 3-dimensional arrays.

9.4. How are 2-dimensional array variables defined? Declare A and B as 5 × 10 matrices. Initialize all their elements to 1 using for-do, while-do and repeat-until constructs.

9.5.(i) Declare PLANET as a linear integer array whose elements can be accessed using EARTH, MERCURY, MARS, MOON, NEPTUNE and SATURN as subscripts.

(ii) Define BIRDS as a two dimensional character array whose elements can be accessed using WATER, GROUND and AIR for the first subscript and SPARROW, CROW, PEACOCOK, HEN, COCK, PENGUIN, DUCK as the second subscript.

9.6. Explain the concept of compatible arrays. Give examples and their uses.

9.7. How is packed array different from unpacked array? Give the relative advantages of the two. Describe the action of the procedures **pack** and **unpack.**

9.8. Give the different ways of defining character strings. Illustrate by examples.

9.9. What do you understand by String Processing? Explain the various operations (by examples) and their implementation in Pascal.

9.10. Design a program to find the complete collating sequence of various characters in the character set of Pascal as available on your system

9.11. Write a program to compute the multiplication table for the numbers 1 to 10 using a 2-dimensional array. Print your results in a tabular form.

9.12. Develop a program which accepts the data

```
NATIONAL PHYSICAL LABORATORY
```

and gives the output as N.P.L.

9.25. Write a program that will read a sequence of integer numbers and finds whether it contains any duplicates.

9.26. Develop a program which prepares the following pattern :

```
          1
       2  *  2
    3  *  *  *  3
 4  *  *  *  *  *  4
    3  *  *  *  3
       2  *  2
          1
```

The character which is to appear at the farthest point of the figure, as 4 in the above pattern, should be given as the input value.

9.27. Consider the differential equation

$$\frac{dy}{dx} = f(x, y)$$

with the initial condition $y(x_0) = y_0$. Choose an interval h, which is sufficiently small, and construct a set of equal spaced points $x_i = x_0 + ih$ (i=0, 1, 2. . . . .). The solution of Eq. (1) $y(x_i) = y_i$, can be obtained in a simple way as

$$y_{i+1} = y_i + h\,f(x_i, y_i)$$

This is known as Euler's method to solve the differential equation.

Develop a flowchart and a program to compute the solution of the differential equation

$$\frac{dy}{dx} = x^2 + y^2$$

with $y(0) = 0$, over the interval (0, 1). Choose the step size in such a way that the solution is obtained to an accuracy of 10^{-6}.

9.28. Design an algorithm and an efficient program to find the real root of the equation

$$1 - x + \frac{x^2}{(2!)^2} - \frac{x^3}{(3!)^2} + \frac{x^4}{(4!)^2} - \frac{x^5}{(5!)^2} = 0$$

by the method of iterations correct to seven decimal places. Print the number of iterations after which the solution converges. Write the program using and without using arrays. Which code is better in your opinion?

(Hint: Rewrite the given equation as

$$x = 1 + \frac{x^2}{(2!)^2} - \frac{x^3}{(3!)^2} + \frac{x^4}{(4!)^2} - \frac{x^5}{(5!)^2} \qquad (2)$$

Neglecting all powers of x, higher than first, we find an approximate value of x to be 1. Assume this as the starting value of x, substitute it on the right-side of Eq. (2). This gives a next approximation to x, say it is $x^{(1)}$. Substitute $x^{(1)}$ in Eq. (2) on r.h.s. and get the new approximation, say, it is $x^{(2)}$. This way continue and obtain $x^{(3)}$ . . . . .,$x^{(n)}$ till $x^{(n-1)} \simeq x^{(n)}$ correct to seven decimal places).

9.29. The co-efficient of rank correlation is defined as

$$r = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n(n^2 - 1)}$$

where d_i = rank difference and n = number of data values. Assume that the following observations are made about 10 students as regards their rank in a Physics class.

Written paper	:	10	7	9	7	8	6	4	2	9	9
Pracitcal paper	:	8	5	7	9	8	8	2	1	8	10

The rank achievements of the students must lie between 1 and 10. Moreover, the number of students cannot be 0 or 1.

Develop a general program to compute the co-efficient of rank correlation taking into account the appropriate constraints.

9.30. Read Programs (a) and (b) carefully, Program (b) uses arrays, while (a) does not.

```
(a)  program COMPARE1 (output) ;
     const N=100 ;
     var I,J,K, S : integer ;
     begin
       S := 0 ;
       for I=1 to N do
         for J = 1 to I do
           for K = 1 to J do
             S : = S+K div 100 ;
       writeln ('SUM=', S)
     end.
(b)  program COMPARE2 (output) ;
     cost N = 100 ;
     var I, J, K, S : integer ;
         A : array [1 . . N, 1 . . N, 1 . . N] of integer ;
       begin
         S : = 0 ;
           for I = 1 to N do
             for J = 1 to I do
               for K = 1 to J do
                 begin
                   A [I, J, K] := K div 100 ;
                   S : = S + A [I, J, K]
                 end ;
               writeln ('SUM =', S)
         end.
```

Run programs COMPARE1 and COMPARE2 on your system and note the execution times in each case. Next, rewrite programs (a) and (b) using **while-do** and **repeat-until** constructs. Again record the execution times. If resources permit, redo the entire experiment for N=150, 200.
Record your time of observations in a tabular form as :

N	for-do		while-do		repeat-until	
	(a)	(b)	(a)	(b)	(a)	(b)
100						
150						
200						

What are your conclusions as regards efficiency of the different versions of the programs with and without arrays? (N.B. if there are any limitations on the choice of value of N, you may try your own values).

Chapter 10

Subprograms: Functions and Procedures

We have seen how declarations, expressions and various kinds of statements are combined to design a computer program. Many times, it may be desired to express a program into smaller units called Subprograms. A subprogram is a program unit/part which performs a particular task. It has its own labels, constants, variables and statements. These are local to the subprogram and have no connection with those appearing in any other program/subprogram. The smaller program units, that is, subprograms may be developed with relatively lesser debugging effort and ease. There is more clarity in their design. Subprograms may be combined to form larger programs. This is referred to Modular Design of programs. A subprogram acts as a module. Libraries of subprograms may be prepared. These program units may be used by others and thereby save program writing time and effort. Moreover, subprograms are written by experienced programmers and thus are designed in an efficient way.

A subprogram may be invoked by a subprogram/program which is called the Calling program. A subprogram is entered at the beginning and after completing the execution of the subprogram, control goes back to the calling program. Also more than one calling program may use the same subprogram. We may depict the invoking of a subprogram by a schematic diagram as shown in Fig. 10.1.

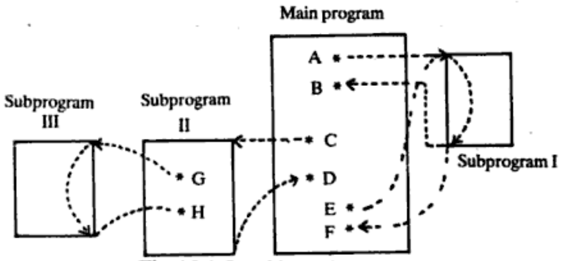


Fig. 10.1: Invoking subprograms

Subprogram I is referred to in the Main program. Subprogram I is invoked at point A. After the execution of subprogram I, control goes back to the Main program at point B immediately after A (or to the point A itself, depending on the type of the subprogram invoked). At point C, subprogram II is invoked which further invokes the subprogram III at point G. After the subprogram III has been executed, control returns to subprogram II at point H and after completing the execution of the remaining part of subprogram II, control comes to point D. Now subprogram II became the calling program for subprogram III. Subprogram I is further invoked by the Main program at point E.

Pascal allows all possibilities of invoking subprograms. For example, subprograms II may invoke I, ~~subprogram~~ I may invoke II or III, and so on. When a subprogram invokes itself, it is called Recursive Call and the process is referred to as Recursion (Section 10.8).

The following kinds of subprograms are available in Pascal:

(a) Function subprograms
(b) Procedure subprograms

We shall discuss the rules to design and use these subprograms in developing programs in the following sections.

10.1 Functions

A function is an independent unit in a Pascal program. It has its own declarations and execution part. The execution part is also referred to as the body of the function subprogram. The format of a function subprogram is:

```
function name (argument-list-1 : t1 ; arguments-list-2 : t2 ; ......) : t ;


local declarations
begin          ─┐
   S1 ;         │
   S2 ;         │
   .            │
   .            │
name: = an expression; │ body
   .            │
   .            │
   .            │
   Sn           │
end;           ─┘
```

Here

name	is an identifier which is the name of the function and is designed according to the Pascal rules to construct identifiers.
argument-list-1, *argument-list*-2, ...	• indicates identifiers separated from each other by comma also called Formal parameters or identifiers • use of formal parameters is optional

	• are the types which may be standard data types or user-defined or subrange type; may be same or different.
	• is the type (may be standard, user-defined scalar or subrange) which applies to the name of the function; it should never be of structured type.
	• type of function name may be same or different from the type of the arguments.
local declarations	• refer to declarations (**label, const, type, var, function, procedure**) which are applicable in the body of function only
$S_1, S_2, \ldots, S_n$	• statements, may be simple or compound

The body of the function is the place where all computations are done. It may consist of any valid statement (simple or compound). There must be an assignment statement of the type.

name : = *an expression*

in the body which assigns a value to the function *name*. This value is returned as the result of execution of the function subprogram upon completion of the function. The last statement in the body must be an **end** statement.

The formal parameter *argument-list* appearing in the function statement may be

- simple or subscripted variables
- array names (or of other structured data type)
- subprogram names
- but never constants, labels and type identifiers.

An example of function statement or function heading is :

function POLYNOMIAL (A, B: **real;** J, K: **integer**) : **real**

Here POLYNOMIAL is the name of the function; A, B, J, K are the formal parameters (A, B are of type real and J, K are of type integer). The type of name POLYNOMIAL is real and it can assume real value. Some more examples of function statement are:

function TEXT (P, L, T : **char;** LAG: **boolean;** M, N : 1 .. 13) **char;**
function CHECKLIST : **boolean** ← (no formal parameters)

Now consider

function PAGE (X, Y : TABLE) : **real;**

where TABLE may have the type defined as

type
  TABLE : **packed array** [1 .. 25] **of char;**

Now the formal parameters X, Y are array names. The use of subprogram name is allowed as a formal parameter. An example is

```
function FORMAT (TEXT : char) : boolean ;
```

Recall that TEXT was the name of the function (defined above) and is being used as a formal parameter where its type is specified. (see Section 10.7).

The formal parameters specify variables (along with their type) which constitute the input to the function subprogram. There can be any number of formal parameters of the type explained above, including none at all.

Let us see the design of a function subprogram. Suppose, we wish to compute A * B * C * D/4.0. A function subprogram for this may be written as:

```
function PRODUCT : real; { No formal parameters with the function
   name PRODUCT }
   const
   FOUR : = 4.0 ;
   var
      A, B, C, D : real;
   begin
      readln (A, B, C, D);
      PRODUCT : = (A * B * C * D)/FOUR;
      writeln (PRODUCT)
   end { function PRODUCT } ;
```

Here no formal parameters have been indicated with the function name PRODUCT. Values of variables A, B, C, D are read in the subprogram body and result is assigned to PRODUCT and then written. The reader should note that the body of a subprogram must have the word **end**, followed by a semicolon (and not period), as the last keyword.

As the subprogram is an independent unit (with its own declarations, type, body, etc) then how does it establish link with the calling program unit? This is done via formal parameters. These parameters also serve to supply input values to the body of the subprogram. Values are passed on to a subprogram via formal paramters when it is invoked by the calling program. This concept is explained in the following section.

Invoking a function subprogram

When a function subprogram is invoked in a statement of a calling program unit, arguments appearing with the name of the function, are called the Actual arguments.

An actual argument may be

- a constant
- a simple or subscripted variable
- an expression
- a subprogram name

appear after the declaration **var.** The complete structure of a Pascal program may be shown schematically as given in Fig. 10.2.

```
program ......(......); { start and the name of the main program }
    label ....;                          ┐
    const ....;                          │
    type .....;                          │  main program declarations
    var .....;                           │
    function/procedure .....             ┘
    { subprogram-1 }
        label ....;                      ┐
        const ....;                      │
        type .....,                      │  subprogram-1 declarations
        var .....;                       │
        function/procedure ....;         ┘
        { subprogram-2 }
            label ....;                  ┐
            const ....;                  │
            type .....;                  │  subprogram-2 declarations
            var .....;                   │
            function/procedure ....;─────┘
            { subprogram-3 }
               :
               :
               :
             begin ─┐
             ......  │  body of subprogram-3
             ......  │
             end ; ─┘

        begin     ─┐
        .........   │  body of subprogram-2
        .........   │
        end;      ─┘

    begin  ─┐
    .....   │  body of subprogram-1
    .....   │
    end ;  ─┘

begin    ─┐
.....     │  body of main program
.....     │
end.     ─┘
```

Fig. 10.2: Common Structure of a Pascal Program

```
  I, J, K, NEXTWORD, LENGTH, LSUB, COUNT, POSITION : integer;
  EQUAL : boolean;
  WORD : packed array [1 .. 10] of char ;
  SENTENCE : packed array [1 .. 100] of char ;
begin
  writeln (' Enter the sentence ending it with a *');
  writeln ;
  writeln (' The sentence is');
  POSITION : = 0 ;
  EQUAL : true ;
  read (LETTER) :
  NEXTWORD : = 1;
  SENTENCE [1] : = LETTER;
  J : = 2;
  LENGTH : = 1;
  COUNT : = 0;
  while LETTER < > '*' do
     begin
        read (LETTER) ;
        SENTENCE [J] : = LETTER ;
        J : = J + 1
     end ;
  I : = 1;
  writeln;
  writeln ('Enter the word to be searched ending it with a blank');
  writeln ;
  read (SUBSTRING);
  repeat
    WORD [I] : = SUBSTRING;
    I : = I + 1;
    read (SUBSTRING)
  until SUBSTRING = '   ';
  LSUB : = I-1;
  writeln;
  J : = 2;
  while SENTENCE [J] < > '*' do
  begin
  if (SENTENCE [J-1] = '   ' and (SENTENCE [J] < > '   ') then
    NEXTWORD : = J ;
    if POSITION = 0 then
    begin
      if (SENTENCE [J] = '   ') or (SENTENCE [J] = '*') then
      begin
      COUNT : = COUNT + 1;
      LENGTH : = J-NEXTWORD;
      if LSUB = LENGTH then
```

```
          begin
            I := 1;
            K := NEXTWORD;
            while (I <= LSUB) and (EQUAL = true) do
            begin
              if WORD [I] = SENTENCE [K] then
                begin
                  I := I + 1;
                  K := K + 1
                end
              else
                  EQUAL := false
            end;
            if EQUAL = true then
              POSITION := COUNT;
            EQUAL := true
          end
        end
      end;
      J := J + 1
      end;
      writeln;
      INDEX := POSITION;
    end; { of function INDEX }
    begin { of main program }
      X := INDEX;
      writeln;
      if X = 0 then
        writeln ('The given word is not in the sentence')
      else
        writeln ('The given word is present in the sentence at position')
      end.
```

Sample input/output

```
Enter the sentence ending it with a *

The sentence is

STRONGLY TYPED LANGUAGES ALLOW TYPE CORRECTNESS
TO BE CHECKED AT TRANSLATION TIME *

Enter the word to be searched ending it with a blank

TYPE

The given word is present in the sentence at position 5
```

```
var
  S : real ;

begin
  S : = (X + Y + Z)/2.0 ;
  AREA : = sqrt (S * (S-X) * (S-Y) * (S-Z) )
end; { procedure ends here }
```

The area is calculated and stored in variable AREA. The procedure TRIAREA can be invoked in a program by indicating its name and the actual arguments. A complete program which invokes this procedure TRIAREA is given below:

Example 10.2

Read data for the three sides of a triangle. Examine the validity of the data read (that is, magnitude of sides should be such that a triangle is formed) and compute the area of the triangle. Make use of procedure TRIAREA discussed above.

A triangle is not formed when any of its side is zero or when sum of any two sides is less than the third side. We shall use this criterian to examine the validity of the input data. In the following program, data are read in the main program and above test is applied.

Program 10.2

```
program AREATRY (input, output);
  var
   A, B, C, D : real ;
  TEST : char ;
  procedure TRIAREA (X, Y,  Z : real ; var AREA : real) ;
                          ↑
                  formal parameters

    var
      S : real;

    begin
      S : = (X + Y + Z)/2.0;
      AREA : = sqrt  (S * (S-X)*  (S-Y) * (S-Z) )
      end ; { of procedure TRIAREA }

  begin { main program }
 repeat
   writeln ('Enter data for the sides);
   read (A, B, C);
   if ( (A = 0.0)  or  (B = 0.0)  or  (C = 0.0) )  then
      begin writeln : writeln ('Invalid data') end
   else
     begin
       if  A > (B + C) or  B > (C + A) or
           C > (A + B)   then
```

```
                    begin writeln ; writeln ('Invalid data') end
                else { invoke procedure }
                    TRIAREA (A, B, C, D) : { procedure TRIAREA invoked }
                    writeln ;
                     writeln ('Area of triangle = ', D : 8 : 4)
        end
        until A * B = sqr (C) ; { program stops when A = B = C } ;
            writeln ('Stop')
        end { program AREATRY }.
```

Sample input/output

```
Enter data for the sides
43.0 34.0 23.0
Area of triangle = 388.8444

Enter data for the sides
23.0 0.0 12.0
Invalid data

Enter data for the sides
3.0 4.0 5.0
Area of triangle = ƀƀ6.0000
Enter data for the sides
3.0 3.0 3.0

Area of triangle = ƀƀ3.8971
Stop
```

The above examples illustrate the specification and use of functions and procedures in complete programs. However, there are certain differences between these subprograms which are listed below.

- Function is invoked by specifying its name in an expression, whereas a procedure is invoked by means of a single independent statement.
- The function name acts-like a variable which can be assigned a value and used in expressions/statements, while the procedure name is merely an identifier and does not have any value.
- Type of the function name is always declared while this is not so for the procedure name.

10.3 Block Structure

We have seen that the format of a general Pascal program is as

```
program ......(.....,.....,....) ;
D₁; D₂; D₃; ......┐
begin             │Block
S₁; S₂; S₃;  ......│
end.              ┘
```

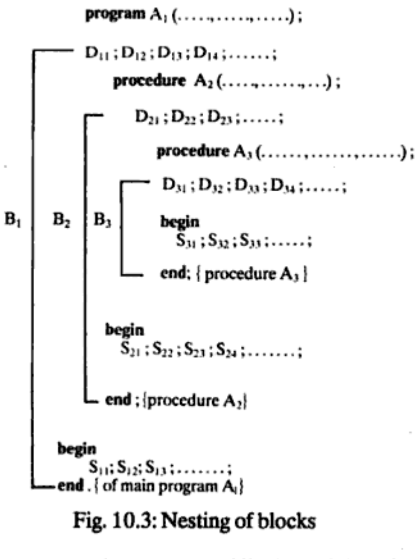
where D_1, D_2, D_3, ... are the declarations and S_1, S_2, S_3, ... are the statements which constitute the body of the program. The body of a program is always enclosed between the reserved words **begin** and **end.**

A Block is defined as consisting of declarations part and body section. This is shown above.

A program may consist of subprograms (functions and procedures) which have their own declarations and body sections. For instance, look at the following example :

```
procedure name (arguments);
      ┌─declarations
      │ begin ─┐
      │ .....  │
Block │ .....  │ ←─procedure-body
      │ .....  │
      └─end ; ─┘
```

In this case, declarations and procedure-body constitute a Block. As subprograms are placed in the declaration part of a main program, so the block corresponding to a subprogram will be contained in the block of the main program. Moreover, a subprogram may contain another subprogram in its declaration part, and so on. This is further brought out by Fig. 10.3.



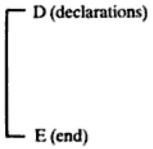
```
          program A1 (....,....,....);
  ┌────── D11 ; D12 ; D13 ; D14 ; ......;
  │         procedure A2 (....,.......,...);
  │   ┌──     D21 ; D22 ; D23 ; .....;
  │   │         procedure A3 (.....,..........,.......);
  │   │   ┌──     D31 ; D32 ; D33 ; D34 ; .....;
B1│ B2│ B3│       begin
  │   │   │         S31 ; S32 ; S33 ; .....;
  │   │   └──     end; { procedure A3 }
  │   │         begin
  │   │           S21 ; S22 ; S23 ; S24 ; .......;
  │   └──       end ; {procedure A2}
  │       begin
  │         S11; S12; S13 ; .......;
  └───────end . { of main program A1}
```

Fig. 10.3: Nesting of blocks

Here A_1 refers to the main program, while A_2 and A_3 refer to the procedures. Procedure A_3 is defined in procedure A_2, while A_2 is defined in main program

A_1. In other words A_3 is contained in A_2 which is further contained in A_1. This is called Nesting of subprograms. Here B_1, B_2 and B_3 are blocks. Block B_3 is contained in block B_2 while block B_2 is enclosed within block B_1. Or, in other words, B_1 is the outermost block and B_3 is the innermost block. Each of the blocks B_1, B_2 and B_3 has its own declarations and statements. These are: D_{11}, D_{12}, D_{13}, D_{14}, ... and S_{11}, S_{12}, S_{13} . . . for B_1; D_{21}, D_{22}, D_{23}, D_{24}, . . . and S_{21}, S_{22}, S_{23}, S_{24} . . . for B_2 and D_{31}, D_{32}, D_{33}, D_{34} . . . . and S_{31}, S_{32}, S_{33}, ... for block B_3.

We donote a block by a square bracket as



where D indicates the beginning of declarations part of a block and E is its end. With this notation, the structure of a Pascal program may be represented in terms of blocks as shown in Fig. 10.4.

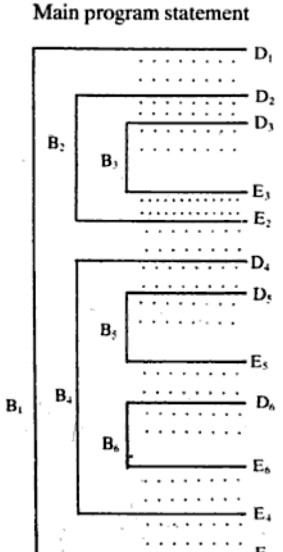


Fig. 10.4: Block structure of Pascal program

or a variation of this.

10.4 Local and Global Identifiers

Identifiers (labels, constants, type, variables, procedures/functions) declared in a block are said to belong to that block and are referred to as Local identifiers. Values of local identifiers are available only in that block and not outside it. Identifiers declared in the declarations of the main program block, that is, outermost block, are called Global identifiers. Their values are available throughout the program and in every block of the program.

Consider the following simple program outline:

```
    program VOLUME (input, output) ;
 ┌── var
 │     VOL, X, Y, Z : real ;
 │     function FIG : integer ;
 │
 │       var ─────────────────────┐
 │       A, B, C :integer ;       │
 │       . . . . . . . . . .      │
 │       . . . . . . . . . .    B2│
 B1      . . . . . . . . . .      │
 │       begin                    │
 │       FIG : = . . . . .        │
 │       end ; { function FIG } ──┘
 │   begin
 │     readln (X, Y, Z) ;
 │     VOL : = FIG (X, Y, Z) ;
 │     writeln (VOL)
 └── end. { program VOLUME }
```

In this program, B_2 is the inner and B_1 is the outer block. Variables A, B, C are declared in B_2. while VOL, X, Y, Z have been declared in block B_1. Variables A, B, C are the local variables while VOL, X,Y, Z are the global variables.

Names of local and global identifiers may be same or different. Let us consider two blocks B_1 and B_2 as shown in Fig. 10.5. B_2 is subordinate to B_1. Block B_1 contains declarations of variables A, B, C and P, Q, T ; while block B_2 has declarations for other variables. These variables may be same as declared in B_1 or different. Suppose the variables declared in B_2 are : P, Q, T, and X, Y, Z and the arrangement appears as shown in Fig. 10.5.

Variables P, Q, T and X, Y, Z are the local variables of block B_2. These variables can be used in statements defined in B_2 only. In addition to this, a statement in block B_2 can also use the variables of block B_1, that is A, B, C. Variables like A, B, C are non local to B_2.

Statements of B_1, which lie outside block B_2, cannot use the variables Y, X, Z. This is because the variables of the inner block are not defined in the outer block. Furthermore, the variables P, Q, T which are declared in B_2 have no connection with the variables P, Q, T declared in B_1. Different memory spaces are assigned to all of them. The variables P, Q, T of block B_1 are not valid in B_2

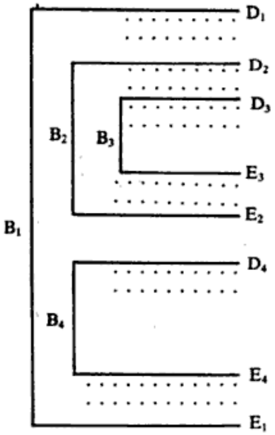


Fig. 10.7: Nested and parallel blocks

Identifier Scope Rules

- An identifier is available only in that block in which it is declared and the blocks which are further contained in it.
- An identifier of an inner block is not defined in the outer block.
- The identifier of parallel blocks are not available to one other.
- Identifiers declared in different blocks may have the same name, but they are all treated as independent.
- Once the statements of a block have been executed and block exited, all identifiers declared inside the block cease to exist. The computer memory is released and can be used for other purposes.
- Identifiers of a block are assigned storage only at the time of execution of the block.
- The same identifier should not be used to denote two different quantities (as for instance, a variable and a constant) within the same block. However, it is permitted to use same identifier for different quantities in different blocks. It may stand for a simple variable in one block, an array in second block, a constant in third block, and so on. Declarations may be same or different.

The nesting of blocks may be continued to any level. However, there is little to be gained from complicating the block structure of a program. It is advisable to keep the nesting to two/three levels

10.5 Value and Variable Parameters

Correspondence between the formal and actual parameters may be established via two mechanisms—commonly referred to as Call-by-Value and

Call-by-Reference. In call-by-value, values of actual parameters are transferred to the formal variables, whereas in call-by-reference, it is the address of actual parameter which also becomes the address of the formal parameter. Each of these modes of information exchange between the subprograms and calling program unit has been implemented in Pascal and are explained below .

Call-by-Value

We have seen that when a function/procedure is invoked, there is established a correspondence between the formal and actual parameters. A temporary storage location is created where the value of the actual parameter is stored. The formal parameter picks up its value from this storage area. This mechanism of data transfer, between the actual and formal parameters, allows the actual parameters to be an expression, functions, arrays, records, files, sets, etc. Such parameters are called Value parameters and mechanism of data transfer is referred to as Call-by-Value. In our discussion so far, we have been considering this type of parameter correspondence. *The corresponding formal parameter represents a local variable in the called subprogram.* The current value of the corresponding actual parameter becomes the initial value of the formal parameter. The value of formal parameter may then be changed in the body of the subprogram by assignment or input statements. This will not change the value of the actual parameter. It is explained by the following example.

```
program CORRESP (output) ;
  var X, Y : integer ;
    procedure PARAM (A, B : integer) ;
      begin
        writeln (A, B) ;                              (1)
        B := A + B                                    (2)
        writeln (B)                                   (3)
      end ; { procedure PARM }
  begin
    X := 3 ;
    Y := 4 ;
    PARAM (X, Y) ;                                    (4)
    writeln (X, Y)                                    (5)
    end. { program CORRES }
```

Here the procedure PARAM is invoked in statement (4) and correspondence between the actual parameters, X, Y and formal parameters A and B is established. Parameter A is initialized to value 3 (initial value of actual parameter X) and B to the value 4. The **writeln** statement (1) writes the values of A and B as

```
3    4
```

Both kinds of parameter calls may be used in subprogram arguments. The differences between call-by-value-and call-by-reference are summarized below:

Call-by-value	*Call-by-reference*
* the formal parameter assumes only the value of the actual parameter	* the formal parameter (always proceeded by the declaration **var**) assumes the address of the actual parameter.
* the formal and actual parameters are two distinct variables	* the formal and actual variables are identical, though their names may be different.
* when the value of the formal (or actual parameter is changed, the corresponding value of actual (or formal) parameter is not changed automatically	* the value of the formal (or actual) parameter automatically changes when the value of actual (or formal) parameter is modified.
* do not allow the transfer of values from the subprogram to the calling program	* values may be returned from the subprogram to the calling program part.
* the actual parameter may be any expression (constant, variable, function, or a combination of these) having the appropriate type	* the actual parameter must be a variable.
* can be used for structured data types	* cannot be used with structured data types.

As a general rule, remember the following : use call-by-value, that is, value parameters to pass values into a subprogram and call-by-reference, that is, variable parameters to return the values from the subprogram.

10.6 Arrays as Subprogram Parameters

You have seen that many kinds of arguments may be used with the subprograms. Now we illustrate further the use of arrays as the parameters of subprograms.

Suppose we want to compute the mean of N data values : $x_1, x_2, \ldots\ldots x_N$. The mean is defined as

$$\text{Mean} = \left(\sum_{i=1}^{N} x_i\right)/N$$

The program to compute the mean, using function subprogram, may be as follows :

```
program AVERAGE (input, output) ;
{ Program to compute mean of data }
var
  M : integer ; Y : array [ 1 .. 100 ] of integer ;
  function MEAN (N : integer ; X : array [ 1 .. N] of integer) : real ;
     var
       J, SUM : integer ;
   begin
     SUM := 0 ;
     for J := 1 to N do
     SUM := SUM + X [J] ;
     MEAN := SUM/N
   end ;
begin
    readln (M) ;
    for J := 1 to M do
      readln Y [J] ;
    writeln MEAN (M, Y) { Function MEAN is invoked }
end.
```

Here M and Y are actual parameters. Y is an array.

The formal argument of the function MEAN has been specified as an array, by the array variable X :

X : **array** [1 .. N] **of integer**

in the usual way. Alternatively, we can also specify as

X : VALUETYPE

where VALUETYPE must have been defined earlier as

```
type
    VALUETYPE = array [1 .. N] of integer
```

Similarly, we can use arrays as arguments with procedures.

The arrays used as parameters may be of any allowed dimensions or any type. Moreover, the arrays may be specified as value parameters or variable parameters.

Next we develop a program to add two matrices and illustrate the specification of arrays of varying sizes.

Example 10.3

Let there be two matrices with elements A_{ij} and B_{ij} $(i = 1, \ldots, m, j = 1, \ldots n)$. The sum matrix with elements C_{ij}

$$C_{ij} = A_{ij} + B_{ij}$$

The reader should appreciate that because of restriction (a), the way one procedure passed as a parameter to another, can communicate results back, is via global variables. (?)

Stadard Pascal does not allow standard functions/procedures to be used as argument, though some implementations of Pascal may allow this.

10.8 Recursion

We have seen that a subprogram may call another subprogram. When a subprogram calls itself, it is referred to as Recursive Call and the process is known as Recursion. To understand the concept of recursion, let us study the following examples.

(i) Factorial of a number, n, (denoted as n !) is defined as

$$\begin{aligned} n! &= n\ (n-1)\,! \\ &= n\ (n-1)\,(n-2)\,! \\ &= n\ (n-1)\,(n-2)\,(n-3)\,! \\ &= n\ (n-1)\,(n-2)\,(n-3)\,(n-4)\,! \\ &= n\ (n-1)\,(n-2)\,(n-3)\,(n-4)\,(n-5)\,! \\ &\ \ \vdots \\ &= n\,(n-1)\,(n-2)\,(n-3)\ldots\ldots 1 \end{aligned}$$

with $0! = 1$

Thus, to calculate n !, we should compute (n-1) ! ; to evaluate (n-1) !, we need to compute (n-2) ! , and so on. The method to compute n ! , (n-1) ! or (n-2) ! , ..., is identical. A specific example is

$$\begin{aligned} 4\,! &= 4 \times 3\,! \\ &= 4 \times 3 \times 2\,! \\ &= 4 \times 3 \times 2 \times 1 \end{aligned}$$

The factorial is defined in terms of itself. This is known as Recursive definition.

A function program to compute the factorial of a number N may be written as :

```
function FACTORIAL (N : integer) : integer ;
  begin
    if N := 0 then FACTORIAL := 1
  else
    FACTORIAL := N * FACTORIAL (N-1)
  end ; { function FACTORIAL }
```

The function name FACTORIAL is appearing in the body of the function also, that is, the function is invoking itself.

Another example of recursion is provided by the Fibonacci number sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Each number is the sum of the two preceding numbers, starting with 0 and 1. We can rewrite the above series as

$$\begin{aligned} f_n &= 0 && \text{if } n = 1 \\ f_n &= 1 && \text{if } n = 2 \\ f_n &= f_{n-1} + f_{n-2} && n > 2 \end{aligned}$$

Thus, the computation of the nth ($n > 2$) Fibonacci number, f (n), needs two earlier numbers f_{n-1} and f_{n-2}. Consider $f_5 = f_4 + f_3$.

$$\begin{tikzcd} f_5 = f_4 \arrow[d] & + & f_3 \arrow[d] \\ f_3+f_2 \arrow[d] & & f_2+f_1 \\ f_2+f_1 & & \end{tikzcd}$$

Thus, the procedure to compute any term ($n \geqslant 3$) is identical.

A recursive function to compute the Fibonacci number sequence appears as :

```
function FIB (N : integer) : integer ;
  begin
    if N = 1 then FIB = 0
      else if N = 2 then FIB := 1
           else
             FIB := FIB (N-1) + FIB (N-2) ;
  end ;
```

The function FIB is being invoked twice in the statement

```
FIB := FIB (N-1) + FIB (N-2)
```

As another illustration of recursive subprograms, we develop a complete program to evaluate the plynomial

$$p_n(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + \ldots\ldots a_{n-1}x + a_n \qquad \text{(I)}$$

Here a_0, a_1, a_2, ... are the co-efficients and n gives the degree of the polynomial. The polynomial can be evaluated for a given value of x, provided a_0, a_1, a_2, ... are known.

The polynomial (I) may be rewritten as

$$p_n(x) = x\,p_{n-1}(x) + a_n$$

where

$$\begin{aligned} p_{n-1}(x) &= a_0x^{n-1} + a_1x^{n-2} + \ldots + a_{n-2}x + a_{n-1} \\ &= x\,p_{n-2} + a_{n-1} \end{aligned}$$

and so on. Thus to evaluate $p_n(x)$, we need to compute $p_{n-1}(x)$ and to find $p_{n-1}(x)$ we should calculate $p_{n-2}(x)$, and so on. This is a recursive design of algorithm.

Example 10.4

Develop a program to evaluate the polynomial (I) for $n = 3$, $x = 2.0$ and $a_0 = 1$, $a_1 = 2$, $a_2 = 3$, $a_3 = 4$, $a_4 = 5$, $a_5 = 6$ using the recursive algorithm.

A program to evaluate the given polynomial may be designed as follows:

Program 10.4

```
    program POLY (input, output) ;
    { Computation of a polynomial by recursive algorithm }
    const M = 10 ;
      var
        J , N : integer ;
      X, VALUE : real ;
                A :  array [0 .. M] of real ;
  function P (N : integer ; X : real) : real ;
    begin
      if N = 0 then
        P : = A [0]
        else
          P : = X * P [N-1, X] + A [N]
      end ; { of function }
    begin { main program }
      write ('Enter the degree of the polynomial (integer mode). . . .  ') ;
      readln (N) ;                                                          . . . .
      write ('Enter the value of X (real mode). . . .   ' ;
      readln (X) ;
      writeln ('Enter the values of a0, a1, a2, . . . .  (in real mode)') ;
      for J := 0  to  N  do
        read (A [J] ) ;  writeln ;
      VALUE : = P (N, X) ;
      writeln ('Value of the given polynomial =', VALUE : 8 : 3)
  end.
```

Sample input/output

```
Enter the degree of the polynomial (integer mode) . . . . 5
Enter the value of X (real mode) . . . . 2.0
Enter the values of a0, a1, a2 . . . .  (in real mode)
1.0   2.0   3.0   4.0   5.0   6.0
Value of the given polynomial is =  ƀ 120.000
```

The polynomial can also be evaluated using the technique called Nesting or Horner's rule. This is an iterative method which can be described as follows:

$$b_0 = a_0$$
$$b_{i+1} = x \, . \, b_i + a_{i+1} \qquad i=0, 1, \ldots, n-1$$

Exercises 12

10.1. Complete the following sentences :

(a) A subprogram is a .... which performs a particular task.
(b) Subprograms are the basis for the design of .... programs.
(c) The body of a subprogram is the place where all .... are done.
(d) The actual arguments must correspond in ...., ... and .... to the corresponding formal parameters in the function/procedure statement.
(e) A block is defined as consisting of .... and ....
(f) Identifiers declared in a block are said to be ....
(g) The nesting of blocks may be continued to .... level.
(h) The **var** formal parameter is a .... parameter and its value is .... on exit from the subprogram.
(i) The type of the function name must not be of .... type.
(j) Names of local and global identifiers may be ....
(k) In the Call-by-Value, the actual parameters may have their values as ...., ... or .... type.
(l) The parameters of subprograms, which are themselves specified as parameters, must not be of .... type.

10.2. Tick the correct answers.

(a) The identifiers appearing in a subprogram are related/unrelated to those appearing in another program or subprogram.
(b) Subprograms save/do not save program development effort.
(c) A subprogram can/cannot be compiled independently.
(d) The function name has/ has/does not have a type.
(e) Subprograms in a program must appear after/before the **var** declaration.
(f) The name of a function/procedure always returns a value.
(g) Identifiers declared in the outer most block are local/global identifiers.
(h) The same identifier can/cannot be used to denote two different entities within the same block.
(i) The formal parameter is assigned the address/value of the actual parameter in the case of **var** parameter.
(j) Recursion allows/disallows the self invoking by a subprogram.
(k)The specification of arguments with function and procedure subprogram is required/ optional.

10.3. (a) What are function and procedure subprograms? Write and explain their formats.
(b) Draw the syntax diagrams of functions and procedure declarations.

10.4. What are formal arguments? What correspondence exists between the actual and formal arguments? Explain the difference between Call-by-Value and Call-by-Reference.

10.5. Summarize the rule for specifying arguments in a function. Can a function make use of variables that do not appear as arguments? Support your answer by examples.

10.6. Point out mistakes, if any, in the following and write the correct format :

(a) **function** RAINBOW (A, B : **char** ; x : **integer**) ;
(b) **procedure** CLOUD (P : **boolean** ; **function** (**var** y : **integer**)) ;
(c) **procedure** ROM (AA, BB : **integer**) : **real** ;
(d) **function** XX (YY : **array** (1 .. 10) **of real**) : **packed array** (1.25)
(e) **procedure** PP (Q : MATRIX) ; **of char** ;
(f) **function** TT (R : **integer** ; **var** S : **array** (1 .. 100) **of char**) : **bollean** ;

```
(g) program CROW ;
     ..........
     ..........
     procedure P1 ;
     ..........
     ..........
     ..........
     procedure P2 ;
     ..........
     ..........
     ..........
     end ; (of P1)
     ..........
     ..........
     ..........
     end ; (of P2)
     ..........
     ..........
     ..........
     end.
(h) procedure P3 (COLOUR = (RED, BLUE, ORANGE, GREEN) ) ;
       begin
         label : 5 ;
         var KOLOR : COLOUR ;
         ..........
         ..........
         ..........
         end ;
```

10.7. Consider the following program outline :

```
program PM (input, output) ;
  label : 5, 10 ;
  const
    C1 = 5 ; C2 = 10 ;

  var
    A, B, C : real ;
      function F1 : real ;
      const 15 ;
      var
        B, C, D : real ;
        ..........
        ..........
        ..........
        begin
        ..........
        ..........
        ..........
        end ; { of F1 }
          function F2 : real ;
          var
            C, D, E : real ;
              procedure P ;
              const 25 ;
```

```
                var
                  D, F : real ;
                begin
                ..........
                ..........
                ..........
                end ; { of P }
            begin
            ..........
            ..........
            ..........
            end ; { of F2 }
    begin
    ..........
    ..........
    ..........
    end. (of program PM)
```

Give the scope of labels, constants and variables indicated in this program.

10.8. What is the output of the following :

```
program RESULT (Output) ;
 var
  M : integer ;
  function NUM (J : integer) : integer ;
     begin
       NUM := 1+J+J*J−M
     end :
begin
  M := 13 ;
  for K := 1 to 5 do
  writeln (M, NUM (K) )
end.
```

10.9. Consider the following program :

```
program TALK (output) ;
label : 2, 3, 4 ;
var
   X : real ;
   function RAM (var  Y  :  real) : real ;
     label : 1 ;
   begin
     1 : writeln (Y) ;
        RAM := X + Y ;
        Y := X * X
   end ;
begin
   X := 4.0 ;
   2 : writeln (RAM (X) )
   3 : writeln (X) ;
   4: writeln (Y) ;
end.
```

What will be the output given by **writeln** statements labelled as 1, 2, 3, 4?

10.10. Develop a function subprogram to produce absolute value of a real number.

10.11. Bring out the differences between function and procedure subprograms. Illustrate by examples.

10.12. (i) Explain the concept of a Block in a Pascal program. What are the rules which should be obeyed while using different blocks in a program?
(ii) Describe the utility of Pascal being a block-structured language.
(iii) Explain the difference between compound statement and a block.
(iv) Prepare a syntax diagram of a block.

10.13. Write a function subprogram to compute the effect of NOT (A OR B) operator. This operator is referred to as NOR. Develop the complete program to print the NOR table using this function subprogram. Here A and B are logical qualities, while NOT, OR, NOR are the logical operators.

10.14. Refer to Example 9.1. This program, to integrate a function using Trapezoidal rule, divides the interval into 100 steps. Now develop a program using function procedure to integrate the function till an accuracy of 10^{-6} is obtained by having different step sizes. Print the answer and the step size.

10.15. Prepare a complete interactive program (using function subprogram) to compute the value of

$$e^x = \sum_{k=0}^{n} \frac{x^k}{k!} \qquad (x < 1)$$

upto 6th place of decimal (using function subprogram) for $x = 0.1, 0.2, \ldots 1.0$. How many terms are needed to achieve the given accuracy in each case. Print your result in a tabular form as

```
X        EXP (X)    N
..       ......     ..
..       ......     ..
..       ......     ..
..       ......     ..
```

10.16. Develop a function subprogram to compute χ^2- test (chi square test) value, defined as

$$\chi^2 = ((f_1-g_1)/f_1)^2 + ((f_2-g_2)/f_2)^2 + \ldots\ldots\ldots\ldots + ((f_n-g_n)/f_n)^2$$

where
f_i = observed frequency of occurrence of i th item
g_i = expected frequency of occurrence of i th item from theory.

Use this subprogram to evaluate χ^2 for the following sample data

item:	0	1	2	3	4	5	6	7
Observed frequency :	10	61	198	335	363	219	79	15
expected frequency :	10	70	210	350	350	210	70	10

10.17. Develop a program package, using subprograms, to perform the following matrix operations :
• sum, difference and product of two matrics
• Trace, norm, transpose of any matrix
• inversion of a matrix

Your package should be able to handle matrices of arbitrary size and take care of the necessary conditions which are imposed on any operation. For instance, for matrix multiplication, the number of columns of first matrix must be equal to the number of rows of the second matrix.).

10.18. Write an interactive program using procedure which computes the chord, if any, where two circles intersect.

10.19. Design a recursive function subprogram to find the greatest common divisor of two positive integers using Euclid's algorithm. (Hint : The greatest common divisor (gcd) function is defined as

$$\text{gcd}(j, k) = \left\{ \begin{array}{ll} \text{gcd}(k, j) & \text{if } k > j \\ j & \text{if } k = o \\ \text{gcd}(k, \textbf{mod}(j, k)) & \text{if } k > o \end{array} \right\}$$

10.20. An integer is said to be palindromic if it reads the same forwards as backwards, e.g. 232, 1331, 63436.

For an positive integer, J, the following sequence usually (not necessarily) converges to a planidromic number:

$$J_o = J$$
$$J_n = (J_{n-0} + \hat{J}_{n-1}) \qquad n = 1, 2, 3, \ldots\ldots$$

where $\hat{J}_{n-1}$ is the integer constructed by writing the digits of J_{n-1} in the reverse order. An illustration is:

$$J_0 = 28$$
$$J_1 = 28+82 = 110$$
$$J_2 = 110+011 = 121$$
$$J_3 = 121+121 = 242$$

Design a complete interactive program, using subprograms, which reads an arbitrary integer number and finds out whether it leads to palindromic number sequence. Print all the palindromic numbers which may be generated by the given integer.

10.21. Prepare a program to generate the function of a Binary Full Adder. Print the output for the sum of following binary inputs A, B, C, defined as :

A	B	C	SUM	CARRY
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

10.22. Develop a procedure named CONVERT which can take two arguments, one an integer and other declared as **packed array** [1 . . 10] of **char**, and converts the integer value into a character string. Write a complete program having procedure CONVERT as its component. Run this program for various sample data.

10.23. Design a procedure to reverse the characters of any string which is passed on to it as an argument. Use this procedure in a program to transform

'SITA RAM' to 'RAM SITA'

10.24. The Ackermann's function is defined as

$$a(m,n) = \left\{ \begin{array}{ll} n+1 & \text{if } m=o \\ a(m-1, 1) & \text{if } n=0 \\ a(m-1, a(m, n-1) & \text{otherwise} \end{array} \right\}$$

Design a recursive function subprogram to compute the Ackermann function for m=n=3.

10.25. There may be many ways in which a positive integer number can be represented as the sum of its integer parts. For example, the number 4 can be represented as

$$\begin{aligned} 4 &= 4 \\ &= 3+1 \\ &= 2+1+1 \\ &= 2+2 \\ &= 1+1+1+1 \end{aligned}$$

Similarly

$$\begin{aligned} 5 &= 5 \\ &= 4+1 \\ &= 3+1+1 \\ &= 3+2 \\ &= 2+2+1 \\ &= 2+1+1+1 \\ &= 1+1+1+1+1 \end{aligned}$$

Let us denote the number of partitions of any positive integer n as p(n). Thus, p(4) = 5, p(5) = 7. Develop a Pascal program, using procedure, to determine p(n) for n=1, 2, 3,...... 10.

10.26. There is a conjecture that "every even number larger than 2 can be written as the sum of two prime numbers". For example, 4 = 2+2, 6=3+3, 8=5+3, 10=5+5, 12=5+7, 14=7+7, 16=11+5, ... and so on. Design a Pascal program, using subprograms, that reads a number, N, determines whether it is even or odd, in the range 1 to 100. And then finds the two prime numbers J and K such that N=J+K.

10.27. The values of the hyperbolic sine and cosine may be obtained from

(i) power series expansion as :

$$\sinh(x) = \sum_{K=1}^{\infty} \frac{x^{2k-1}}{(2k-1)!}$$

$$\cosh(x) = \sum_{k=1}^{\infty} \frac{x^{2k}}{(2k)!}$$

where $-\infty < x < \infty$.

(ii) the recurrence relations as

$$\sinh(x) = \sum_{k=1}^{n} u_k$$

$$u_1 = x, \quad u_{k+1} = \frac{x^2}{2k(2k+1)} u_k$$

$$\cosh(x) = \sum_{k=1}^{\infty} v_k$$

$$v_o = 1, \quad v_{k+1} = \frac{x^2}{(2k+1)(2k+2)} v_k$$

Design a Pascal program, using subprograms, to evaluate tanh (1.4) upto sixth place of decimal by the two algorithms (i) and (ii). Print the number of terms needed in the two cases. Plot the sinh and cosh functions. Compare the approximate time needed to obtain the desired accuracy.

10.28. Develop a program that will obtain the value of a complex number raised to a real power. (Hint : make use of de Movier's theorem).

10.29. If p is the probability that an event will take place in any single trial and $q=1-p$ is the probability that it will not take place in any single trial, then the probability that the event will take place exactly m times in n trials, is given by :

$$P(m) = \frac{n!}{m!(n-m)!} p^m q^{n-m} \qquad (I)$$

where m=0, 1, 2, ... , n. Eq. (I) represents a discrete probability distribution and is called the Binomial distribution. P(m) is the binomial distribution function.

Develop an algorithm and a program, using subprograms, to plot BInomial distribution for p=0, 0.1, 0.2, 0.3 . . . . 1.0 and different values of n and m. (If your program does not work for large values of n, compute its value using the relation

$$n! \approx \sqrt{2\pi n}\, n^n e^{-n}$$

10.30. It is said "subprograms enhance program readability and maintainability." Justify this statement.

Chapter 11

Record Data Types

We are all familiar with data. Data are a collection of facts that are generally in an 'unorganized' or 'raw' form. In order that some information can be derived from this, data must be processed. In fact, Data processing implies putting the data into an organized and useful form. We can perform several operations on data, such as recording, storing, retrieving, verification, classification, sorting/merging, and so on. And obviously, computer is the most powerful tool to carry out these operations. It helps to process data at a very fast speed and accurately. Processed data is also referred to as Information. In order that data are easily and efficiently processed, it is organized into the form of Records and Files. Let us first understand the concepts of records and files and then go to their definition and manipulation using the facilities as available in Pascal.

11.1 Record and Files

We know that all quantities in a program are expressed in terms of characters of the Pascal character set. We define a data item as the basic unit of data. It is treated as the primary unit for processing of data. An elementary data item is also known as Field. Examples of data items (fields) are : names (may be of persons, objects, or any other item), addresses (may be of persons, offices, companys, etc.), date of birth, rates, length, invoice number, and so on. Each data item is identified by a unique identifier. A collection of related fields (data items), treated as a unit, is called a Record; whereas a set of related records is referred to as a File. To illustrate these concepts we design a file of students in a class. Let there be 30 students in the class MCA-I. Each student will have its own details. Let us write down the particulars of one student. Say, these are as

Role Number	: 13
Name	: Pauli
Class	: MCA-I
Department	: Computer Science
Telephone	: 2912791

Here Role Number, Name, Class, Department, Telephone are the names of data items or field identifiers. Collection of these five fields forms a Record. A record may be assigned a name. Let us call this record by the name STD-REC. Its organization may appear as

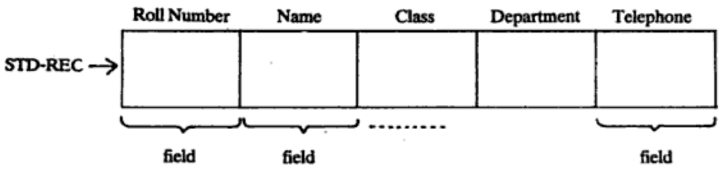


Fig. 11.1: A record

Here 13, PAULI, MCA-I, Computer Science, 2912791 are the values of fields or field contents. The number of characters in the value of a field defines its length, also referred to as Field Width. Thus, width of various fields in the STD-REC are :

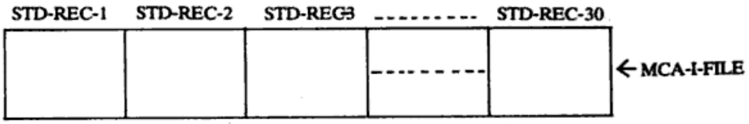
Field Value	*Width*
13	2
PAULI	5
MCA-I	5
COMPUTER ƀ SCIENCE	16
2912791	7

The number of fields in a record defines the size of a record. In the above example, size of the record is 5 fields and occupies a space of 2+5+5+16+7 = 35 characters. Size of a record may be fixed—called Fixed Records or variable-referred to as Variable Records. The arrangement of fields in a record defines its Format. Every record has a particular format. In our example, the format of STD-REC has been defined by Fig. 11.1. The record may consist of any number of fields, minimum being one field.

There is one record for each student. There are 30 students in the class, so there will be 30 records. Let us call these records by the names STD-REC-1, STD-REC-2, ..., STD-REC-30. We know that a collection of related records forms a File. Thus, the records of 30 students constitute a file. In other words, we say that the file of students of MCA-I class has 30 records. Similarly, we can define files for MCA-II class. Let us call the student file of MCA-I class as MCA-I-FILE. In practice, we come across files of several kinds such as inventory file (contains inventory records for items in stock), hospital patient file (contains records of patients in a hospital), sale-transaction file, payroll file, and so on. In fact, each file refers to a set of related records of that particular discipline/area.

In Pascal, there is generally one record format for any particular file; however, different files can have different record formats. Record format is decided by the specific application as to how many and how the fields in a record are going to be included and organized. Files may consist of fixed length or variable length records. With fixed length records, there is need to specify the maximum space for storing data at the start whereas with the variable length length records, only the required space needed by data is specified at the start and not the maximum.(?).

A file is a collection of related records. In other words, records constitute a file or are the components of a file. (A record may consist of a single field even. In that case, components of the file will be fields). All components of a file must be of the same type. Records (components) are organized serially in a file. An illustration is



Records in a file may be reached (more appropriately accessed) either sequentially or randomly. Files which allow only sequential access are the Sequential files whereas those which permit both sequential and random access to its records are known as Random or Direct access files. Files and records play an important role in business data processing applications as it is very convenient to manipulate and process data organized in records and files. All input/output operations are performed using records/files between user programs and the peripheral devices. The data in the form of files can be easily retrieved, updated, deleted or sorted.

Pascal offers facilities to define record and file data types. Both of these are structured data types. In this chapter, we shall study the record data types. their usage and applications and defer the presentation of file data types to the next chapter.

11.2 Record Data Types

We have seen that a record consists of fields. The fields may be of the same or different types, that is, may be numeric or non-numeric. The type of a field may be declared as usual. The record type is defined as follows :

```
type
  record-type-identifier = record
                             f1 : t1 ;
                             f2 : t2 ;
                             ......
                             ......
                             fn : tn
                           end ;
```

where

record-type-identifier	the user-defined type name,
$f_1, f_2, \ldots\ldots\ldots$	field identifiers

$t_1, t_2, \ldots\ldots\ldots$ indicate the type of the fields ; may be standard, enumerated, subrange, arrays or records

Let us refer to the STD-REC-1 defined in Section 11.1. In this record, fields Name and Telephone are numeric (integer type) while all others are non-numeric. In fact, we have taken them to be of character type. A record type consisting of these fields may be defined as

```
type
  STDREC = record
             ROLLNUMBER     : integer ;
             NAME           : packed array [1 .. 15] of char ;
             CLASS          ; packed array [1 .. 5] of char ;
             DEPARTMENT     : packed array [1 .. 15] of char ;
             TELPHONE       : integer
                end ;
```

Here identifier STDREC defines the record-type. The type of each field has also been defined. Fields of identical type can also be declared in a single declaration. The above record type may also be specified as

```
type
  STDREC = record
  ROLLNUMBER, TELEPHONE     : integer ;
  NAME, DEPARTMENT          : packed array [1 .. 15] of char ;
  CLASS                     : packed array [1 .. 5] of char ;
       end ;
```

Another example of defining the record type is

```
type
  DATEOFBIRTH =     record
                    DAY : 1 .. 30 ;
                    MONTH : 1 .. 12 ;
                    YEAR : integer
                    end ;
```

Here the identifier, DATEOFBIRTH, defines record type while DATE, MONTH, YEAR are the fields. The fields DATE and MONTH are of subrange type and the field YEAR is of type integer.

The record variables may be defined in the usual way as

---

var
  $v_1, v_2, \ldots$: *record-type-identifier*

---

where $v_1, v_2, \ldots\ldots$ are variable identifiers.

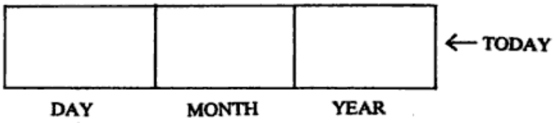
Again refer to our example of STDREC. The identifier STDREC defines a type of a record. Suppose we wish to define variables STDREC1, STDREC2, STDREC3 of this type. It can be done as

```
var
  STDREC1, STDREC2, STDREC3 : STDREC
```

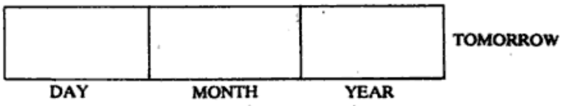
Next, say we want to declare record variables TODAY and TOMORROW of type DATEOFBIRTH. This may be done as

```
var
  TODAY, TOMORROW : DATEOFBIRTH
```

The fields of the variable TODAY are



Similarly, fields of record variable TOMORROW are



Another example is

```
type
  EMPREC = record
    NAME : packed array [1 .. 25] of char ;
    DEPTT : packed array [ 1 .. 10 ] of char ;
    SALARY : real ;
    DUTYDAY : (MON, TUES, WED, THRS, FRI)
        end ;
var
  NAMEX, NAMEY, NAMEZ : EMPREC
```

Here variables NAMEX, NAMEY, NAMEZ are the record variables of type EMPREC. The fields of the records are NAME, DEPTT, SALARY, DUTYDAY, with the types indicated.

NAMEX, NAMEY, NAMEZ have identical record structures with four fields as their components.

Record variables may be defined in a variable declaration directly as well. An example is

```
if (ADDRESS · COLONY = 'VAISHALI' then
  ADDRESS · LOCALITY = 'PITAMPURA')
else
  ADDRESS · LOCALITY = 'SHALIMAR BAGH' ;
writeln (ADDRESS · COLONY, ADDRESS · LOCALITY) ;
```

11.4 Hierarchical Records

The field(s) of a record may be another record(s), whose field (s) may be record(s) further, and so on. This may be indicated as

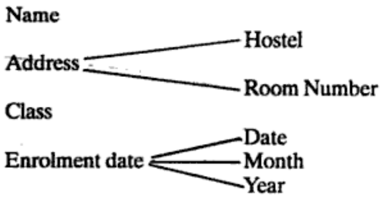
```
type
  record-type-identifier = record
    f1 : t1 ;
    f2 = record
      f21 : t21 ;
      f22 · : t22 ;
      f23 : record
        f31 : t31 ;
        f32 : t32 ;
        .
        .
        .
      end
    end ;
    f3 : t2 ;
    .
    .
  end ;
```

where t's indicate type and f's denote field identifiers. Here f_1, f_2, f_3, ... are fields of record, specified by *record-type-identifier*, in which f_2 is a record. Similarly, f_{21}, f_{22}, f_{23} are the fields of record f_2. Further, f_{23} is also record whose fields are f_{31}, f_{32}, and so on.

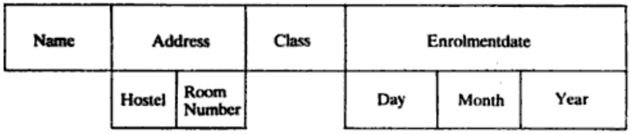
The record variables of *record-type-identifier* can be defined in the usual way as :

```
var
  v1, v2, . . . . .   . . . . . : record-type-identifier
```

These concepts are illustrated by the following examples.
Suppose, details of a student in a class are :



Diagramatically, we can represent the structure of these details as :



This is a Record. Let us name it STUDENTREC. Name, Address, Class and Enrolementdate are the fields of STUDENTREC. Fields Address and Enrolmentdate are records further. A Pascal description of the STUDENTREC may be prepared as :

```
type
  STUDENTREC = record
                    NAME : packed array [1 .. 20]  of char ;
               ADDRESS : record
                         ROOMNO : integer ;
                         HOSTEL : packed array [1 .. 15] of char
                         end ;
                    CLASS : ( I, II, III, IV, V ) ;
 ENROLMENTDATE = record
                         DATE : 1 .. 20 ;
                         MONTH : 1 .. 12 ;
                         YEAR : integer
                         end
                     end ;
```

Further, we can define record variables of type STUDENTREC as :

STUDENT1, STUDENT2, STUDENT3: STUDENTREC ;

Here STUDENT1, STUDENT2, STUDENT3 are record variables of type STUDENTREC. The component fields of these variables are :

```
NAME
ADDRESS (a record)
CLASS
ENROLMENTDATE (a record)
```

Records, as defined above, are referred to as Hierarchical records. Remember in hierarchical records, the fields themselves may be records. This is also referred to as Nesting of Records and is quite useful to define new data structures.

Fields of hierarchical records can be accessed in the same way as discussed earlier by the use of period. Suppose, we wish to refer to the field ROOMNO of record ADDRESS of record variable STUDENT1. This can be done as

```
STUDENT1 . ADDRESS . ROOMNO
```

Similarly, the field MONTH of record ENROLMENTDATE of record variable STUDENT2 may be accessed as

```
STUDENT2 · ENROLMENTDATE · MONTH
```

Same procedure is adopted for other component fields.

Example 11.1

Develop a program which can perform the arithmetic operations—addition, substraction, multiplication and division—with two complex numbers.

Let $A = a_1+ib_1$ and $B = a_2+ib_2$ be two complex numbers. a_1 and a_2 are the real parts while b_1 and b_2 are the imaginary parts. The symbol $i = \sqrt{-1}$. The arithmetic operations for complex numbers are defined as :

$$\begin{aligned} A + B &= a_1 + ib_1 + a_2 + ib_2 \\ &= (a_1 + a_2) + i(b_1 + b_2) \\ A - B &= (a_1 - a_2) + i(b_1 - b_2) \\ A \times B &= (a_1 + ib_1)(a_2 + ib_2) \\ &= (a_1 a_2 - b_1 b_2) + i(a_1 b_2 + b_1 a_2) \end{aligned}$$

$$A \div B = \frac{a_1 + ib_1}{a_2 + ib_2}$$

$$= \frac{a_1 a_2 + b_1 b_2}{a_2^2 + b_2^2} + i\,\frac{b_1 a_2 - a_1 b_2}{a_2^2 + b_2^2}$$

The Pascal program to implement the four complex arithmetic operations may be as follows :

Program 11.1

```
program COMPLEXOPR (input, output) ;
   { program for complex arithmetic operations }

   type
     COMPLEX = record { record is introduced here }
                   A, B : real
                   end ;

   var
     X, Y, Z : COMPLEX ;
     C : real ;
     CH, OPERATOR : char ;

   begin { read the complex numbers }
      writeln ('Enter the complex number : first real part and then imaginary
                                                                      part'),

     readln (X.A, X.B) ;
     readln (Y.A, Y.B) ;

   repeat
     write ('Enter the operator?') ;
     readln (OPERATOR) ;
     case OPERATOR of

  : begin
          Z.A:=X.A+Y.A ;
          Z.B:=X.B+Y.B ;
          writeln ('Sum of complex numbers') ;
          end ;{ + }

'–' : begin
      Z.A:=X.A–Y.A ;
      Z.B:=X.B–Y.B ;
      writeln ('Difference of complex numbers')
      end ; { – }

'*' : begin
      Z.A:=X.A*Y.A–X.B*Y.B ;
      Z.B:=X.A*Y.B+X.B*Y.A ;
      writeln ('Product of complex numbers')
      end ; { * }

'/' : begin
      C:=Y.A*Y.A+Y.B*Y.B ;
      Z.A:=(X.A*Y.A+X.B*Y.B) / C ;
      Z.B:=(X.B*Y.A–X.A*Y.B) / C ;
      writeln ('Division of complex numbers')
    end ; { / }
```

```
var
  COMPLEX1, COMPLEX2, COMPLEX3 : record
                                 REALPART : real ;
                                 IMAGPART : real
                                            end ;
```

Now COMPLEX1, COMPLEX2, COMPLEX3 are shared type record variables; their fields are REALPART and IMAGPART. Suppose, we initialize

```
COMPLEX1 · REALPART : = 13.0 ;
COMPLEX1 · IMAGPART := 3.0 ;
```

then an assignment statement of the type

```
COMPLEX2 : = COMPLEX1 ;
```

will intialize the fields of record COMPLEX2 as

```
COMPLEX2 . REALPART → 13.0
COMPLEX2 . IMAGPART → 3.0
```

Record variables may be used as formal and actual parameters of subprograms, provided they are of shared type. Moreover, their usage may be as value parameters or as variable parameters. These concepts are illustrated by the following examples and implemented in Program 11.2.

11.6 Arrays of Records

We can define an array of records as well. This allows the creation of new data structures. An example is

```
type
  CLASS = record
            LASTNAME : packed array [1 .. 10] of char ;
  FIRSTNAME, MIDNAME : char ;
  TEST1, TEST2, TEST3, PERCENTMARKS : real
          end ;

var
  MUSIC : array [1 .. 50] of CLASS ;
```

Here MUSIC is an array whose each element is a record of type CLASS. The first element of array MUSIC is MUSIC [1]. MUSIC [1] is a record with component fields as : LASTNAME, FIRSTNAME, MIDNAME, TEST1, TEST2, TEST3, PERCENTMARKS. We may represent this as

MUSIC [1]

LASTNAME	FIRSTNAME	MIDNAME	TEST1	TEST2	TEST3	PERCENTMARKS

```
      writeln ('Number of boys in arts =', A[4] ) ;
      writeln ;
      TOTAL := 0 ;
      for I := 1 to 5 do TOTAL := TOTAL + Y[I] . SCI . BOYNUM ;
      A[5] := TOTAL ;
      writeln ('Number of boys in science =', A[5] ) ;
      writeln ;
      TOTAL : = 0 ;
      for I : = 1 to 5 do TOTAL := TOTAL+ Y[I] . COM . BOYNUM ;
      A[6] : = TOTAL ;
      writeln ('Number of boys in commerce = ', A[6]) ;
      writeln ;
      TOTAL : = 0 ;
      for I : = 1 to 5 do TOTAL := TOTAL+ Y[I] . ARTS . GIRLNUM ;
      A[1] : = TOTAL ;
      writeln ('Number of girls in arts =', A[1]) ;
      TOTAL : = 0 ;
      for I : = 1 to 5 do TOTAL := TOTAL + Y[I] . SCI . GIRLNUM ;
      A[2] : = TOTAL ;
      writeln ;
      writeln ('Number of girls in science =', A[2] ) ;
      writeln ;
      TOTAL := 0 ;
      for I := 1 to 5 do TOTAL := TOTAL + Y[I] . COM . GIRLNUM ;
      A[3] : = TOTAL ;
      writeln ('Number of girls in commerce = ', A[3] ) ;
      writeln ;
      for I : = 1 to 35 do write ('-') ;
      writeln ;
      TOTAL := 0 ;
      for I := 1 to 6 do TOTAL := TOTAL + A[I] ;
      writeln ('Total number of students in college =', TOTAL) ;
      writeln ;
      for I := 1 to 35 do write ('-') ;
      writeln
end ;     { end of QUERY2 }
{------------------------------------------}
procedure QUERY3 ;
{ This procedure determines the stream in which there are maximum number of
boys }
{ and the stream in which there are maximum number of girls }
  var BOYMAX, GIRLMAX : integer ;
begin
  FINDMAX (4, 6, A, BOYMAX) ;
  FINDMAX (1, 3, A, GIRLMAX) ;
  case BOYMAX of
```

If we use the **with** statement, then name of the record has not to be specified with each field. It appears only once after the keyword **with.**

The format of **with** statement is

with v **do** S

where v is the name of record type variable and S is a statement which may be simple or compound. S is called the scope of the **with** statement. Pascal allows more general form of **with** statement as :

with $v_1, v_2, v_3 \ldots, v_n$ **do** S

which is equivalent to

```
with v1 do
   with v2 do
      with v3 do
         ........
         ........
      with vn do S
```

Suppose records R1 and R2 use same identifier, say A, for a particular field, and we specify the following nesting :

```
with R1 do
  with R2 do
     A:=B+C;
```

This identifier A is the one associated with record R2. To perform any operation on the field A of record R1, an explicit reference as R2 . A will be required.

The general format is used with nested records. Its use is illustrated by more examples in the following sections.

The use of the **with** statement reduces the length of the program code and increases its readability. It may also produce an efficient program code.

11.8 Variant Records

In many applications, it may be desired to introduce different field(s) in a record depending on some values/conditions, and so its (their) type must be defined accordingly. To illustrate this, let us design a record of a person when the following information is available:

Name
Address (house number, name of colony, city, pincode)
Sex (male, female)
Age (20 . . 60)
Job (emp, unemp, unknown)

Call this record by the name EMPREC. The Pascal record may be defined as:

```
var
  EMPREC = record
              NAME : packed array[1 .. 25] of char ;
                 ADDRESS : record
                 HOUSENUMB : integer ;
                 COLONY, CITY : packed array [1 .. 25] of char ;
                 PINCODE : integer
                           end ;
              SEX : (MALE, FEMALE) ;
              AGE : (20 .. 60)
              JOB : (EMP, UNEMP, UNKNOWN)
              end ;
```

Now, suppose if field JOB is EMP, then we wish to know the name of the employer, but if JOB is UNEMP, then we may like to know since when has the person been unemployed. However, if field JOB is UNKNOWN, then the record may be closed. In order to build this requirement into a record, use of **case** statement is made in place of field JOB as :

```
case  JOB : (EMP, UNEMP, UNKNOWN) of
EMP : (EMPLOYER : packed array [ 1 .. 20 ] of char ;
UNEMP : (LASTEMP : array [1 .. 10] of char) ;
UNKNOWN : (CLOSE : packed array [1 ..5] of char)
  end ;
```

Thus, the complete record EMPREC appears as

```
var
  EMPREC =     record
             ┌─ NAME : packed array [1 .. 25] of char ;
             │  ADDRESS : record
             │  HOUSENUMBER : integer ;
Invariant fields│  COLONY, CITY : packed array [ 1 .. 25 ]  of char ;
             │  PINCODE : integer
             │  end ;
             │  SEX : (MALE, FEMALE) ;
             └─ AGE : (20 .. 60) ;

             ┌─ case  JOB : (EMP, UNEMP, UNKNOWN) of
             │  EMP : (EMPLOYER : packed array [1 .. 20] of char) ;
Variant field │  UNEMP : (LASTEMP : array [ 1 .. 10 ]  of char) ;
             │  UNKNOWN : (CLOSE : packed array [ 1 .. 5 ]  of char)
             └─ end ;
```

See that the record has two parts: fixed (invariant) and variant. The variant part depends on the value of the field such as field JOB in the above example. If the value of field JOB is EMP, then the record has the field named EMPLOYER of type packed array; if the value of the field JOB is UNEMP, then the record has a field called LASTEMP, while for the case when JOB value is UNKNOWN, the field CLOSE will exist in the record. The user must note that the type of each of the new fields EMPLOYER, LASTEMP and CLOSE has been indicated.

The field of a record which is set up using a **case** statement, as given above, is referred to as the Variant field, while the entire record is called a Variant record. Thus, EMPREC is a Variant record.

As another example, let us design a record for a book according to the following information:

Author
Publisher
Year of publication
Price
Status

We wish that the field, Status, should entail the information as:

if the book is sold, then Status is true and we should know the saleprice and month of sale; if the book is not sold, its Status is false.

The record can be designed as

```
      var
         BOOK       =record
                   ┌─ AUTHOR, PUBLISHER: packed array [1 ..20] of char ;
Invariant fields   │  PRICE, YEAR : integer ;
                   └─ STATUS : boolean ;

Field Status        ┌case STATUS of
has been defined    │true : SALEPRICE : integer ;
as of invariant type│      MONTH : integer ;
                    │false : (SOLD : boolean)
                    └end ;
```

Thus, when STATUS is true, two additional fields—SALEPRICE and MONTH will exist while if STATUS is false, only one field SOLD will be there in the record. The reader should note that new fields may be introduced into a record during the program execution via variant part of the record depending on certain conditions, of course.

Fields indicated in the variant part may be accessed in the usual way. If STATUS is **true**, then we can reach the fields SALEPRICE and MONTH as

```
BOOK . SALEPRICE
BOOK . MONTH
```

However, when STATUS is false, then the field, SOLD is part of the record and can be reached as BOOK. SOLD.

The field which appears with a **case** statement is called the Tag field. In our above examples, JOB and STATUS are the tag fields.

The general format of the variant part (field) of the record appears as

```
case tag-field : type of
     tag-value-1 : (field-list-1) ;
     tag-value-2 : (field-list-2) ;
     . . . . . . . . . . . . .
     . . . . . . . . . . . . .
     . . . . . . . . . . . . .
     tag-value-n :(field-list-n) ;
```

where *field-list*-1, *field-list*-2, . . . indicate the newer fields introduced alongwith their type. *The field-list must be enclosed within parentheses. tag-value-1, tag-value-2*, . . . are the values which the tag-field can assume. The tag-field must always be ordinal constants. By examining the tag-field, we are able to determine which data fields have been used, and, hence, the structure of the record.

The use of keyword **case** has different syntax here than in the **case** statement. There is no **end** associated with the present **case** word as the end of the record also serves its end.

It may also be required not to indicate any field after a tag-value. In such a case, specify the parentheses as ( ). Moreover, a record may consist of only a variant part. An example is

```
type
  VEHICLE = (SCOOTER, CAR, BUS, TRUCK, CYCLE, VAN) ;

var
  CONVEYANCE = record

     case LICENSE : VEHICLE of                   ┐
     CYCLE : ( ) { empty }                       │  Variant part
     SCOOTER, CAR : (CATEGORY : 1 .. 2) ;        │  only
     BUS, VAN, TRUCK : (WEIGHT : 100 .. 1000)    │
     end ;                                       ┘
```

Here CONVEYANCE is a record which consists only of the variant part.

The variant part must be specified after specifying the invariant part of the record. However, there can be only one variant field, though variants may be nested.

We have seen that the purpose of the tag-field is to specify a value which indicates the alternative of the variant which is in effect. Pascal allows the optional use of the field. An example is

```
record
  case boolean of
     false : (X : real) ;
     true : (LIMIT : 1 .. 128)
end ;
```

The type **boolean** is choosen as the selector of the **case** because it defines two possible values which is what is required to specify the two alternatives.

In variant records, all fields of the variant part do not automatically become component of the record. The variant specified by the **case** constant becomes the part of the record. This is the Active while all others are the Inactive variants. The values of inactive variants are not defined. It is an error to attempt to use an inactive variant.

The total storage occupied by a record is the size of the fixed part plus its longest variant part.

We explain this concept and the reference to tag-field of variant records by the following example.

```
type
      FIGURE = (SQUARE, RECTANGLE, TRIANGLE) ;
GEOFIGURE  = record
               X, Y : real ;
          case FIG : FIGURE of
                     SQUARE : (LENGTH : real) ;
                     RECTANGLE : (LENGTH, BREADTH : real) ;
                     TRIANGLE : (SIDEA, SIDEB, SIDEC : real)
                     end ;

          var  FIGVAR : GEOFIGURE ;
```

Now FIGVAR is a record variable of type GEOFIGURE. This record definition contains a fixed part as well as a variant part with tag-field FIG.

The storage allocated is for the two real variables X, Y, of the fixed part of the record, the tag-field FIG and the three variables SIDEA, SIDEB, SIDEC of the field TRIANGLE (the largest field.) The other fields SQUARE and RECTANGLE require less storage than this field. The arrangement of storage allocation for record variable FIGVAR may appear as showin in Fig. 11.2(a)

The tag-field can also be referenced in the same way as other fields. An example is FIGVAR . FIG. List the values that can be assigned to this variable in the above example.

It is possible that the types of fields in the variant part of the record are different. For instance, consider the example on Page 238 of variant record BOOK. Now the arrangement of storage allocation appears as given in Fig. 11.2(b)

Pascal does not permit reading a record from input unit nor writing a record on the output unit. Only fields of the record may be read/written provided these operations are defined for the field. For example, we can neither read nor write fields of an enumerated type.

11.9 Packed Records

The data constituting a record may also be stored in the computer memory in the packed mode. This may be done by using the reserved word **packed** before the word record as illustrated below :

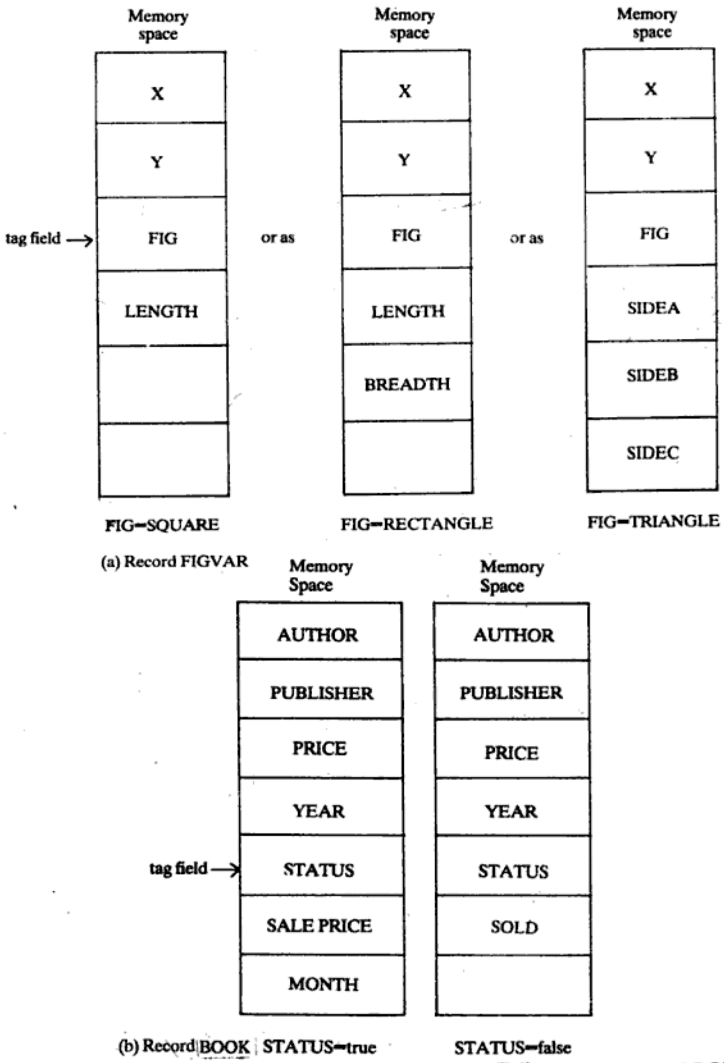


Fig. 11.2: Storage allocation for (a) record variable FIGVAR, and (b) record BOOK

```
record-type identifier = packed record
                            f1 : t1 ;
                            f2 : t2 ;
                            .......
                            .......
                            .......
                         end ;
```

The system will automatically allot space to the various fields f_1, f_2, ... in the packed mode. Other concepts, as discussed with packed arrays, are applicable to packed records as well.

11.10 Differences between Arrays and Records

Arrays and records are structured data types. Both refer to a collection of elements. However, there are certain differences between them and are given below.

* all components (elements) of an array are of identical type, while components (fields) of a record may be of identical or different types.
* components of an array are referenced by indices which may be constants, variables or expressions, whereas the fields of a record are referenced by names as:

  record-variable-name . field-name

* a record may be of variant type, while there is no such concept associated with arrays.
* arrays may be of 1-, 2-, 3-, ... dimensions whereas this is not applicable to records.
* the **with** statement is defined for **record** data structure, while there is no such statement for array data structure.

After having studied records structure, next we go over to the study of file data types.

Exercises 11

11.1. Tick the correct answers in the following sentences.

(a) Files are/are not useful for data processing.
(b) A record consists of fields which may/maynot be of different types.
(c) Scalar fields of a record can/cannot be assigned values by an input statement.
(d) Shared type records are of same/dissimilar type.
(e) Actual parameters of a subprogram may/may not be of record type.
(f) Fields of a record may be simple/structure/both type of variables.
(g) The **with** statement can/cannot be followed by a compound statement.

11.12. Bring out the significance and role of a tag-field in a variant record. Can this field be omitted? Suppose it is so, what difficulties do you envisage?

11.13. Compare and contrast the structured data type arrays and records. Give examples in support of your answer.

11.14. Prepare a record structure for a triangle which consists of the following fields

SIDEA, SIDEB, SIDEC, ANGLE

Call this record TRIRECORD. Using this record and for the given data values of the sides and angles, develop a program to find the type of the triangle.

11.15. Using the record TRIRECORD, write a procedure that accepts as input the record variable and returns the type of the triangle. Then prepare a complete program and test it for a sample data.

11.16. Develop a program, using records, that converts cartesian co-ordinates to polar co-ordinates and vice versa.

11.17. Define two arrays A and B of compatible records. Design a program which deletes a record from array A and appends it to array B.

Chapter 12

File Types

We learnt about files in the previous chapter. Pascal supports only sequential files. The components in such files are arranged in a serial order, one after another. Programs communicate with each other and outside environment via files. You will recall that in the statement, **program**, arguments are as **input, output** or names of some other files. These serve to establish a link between the program and external media. **input** and **output** are the standard file names. We can define our own files as well.

Pascal treats the incoming data to a program from input devices (terminals, tapes, disks, etc.) or outgoing data from a program to the output devices, as files. All these devices are different, but Pascal deals with them as of the same kind of logical devices. This means that though the input/output devices are physically dissimilar, but no distinction need be made in the **read/write** statements in a program. The Pascal system will automatically establish contact with the appropriate device as indicated by the system commands. Moreover, you know that data are represented differently on different devices. For instance, data on cards, terminals or tapes, etc., have different physical or external representations. When data are read by the Pascal system, they are transformed into the internal, also called Logical representation. Similarly, data in the internal form are changed to the external form, when results are to be output, by the Pascal system. Such conversions are affected automatically by the system and the user is not concerned with this.

Data resident on external devices form the external files, while those inside the computer memory, constitute internal files. Names can be assigned to the files and referenced. Access to components of external files is rather slow as they are stored externally, but access to internal files is extremely fast (!).

A file, that is created outside the Pascal program, can be passed on into the program by specifying the name of the file as a parameter with the **program** statement. Similarly, a file which is created in a program (internal file), can be passed on to the outside environment by indicating its name in the parameter list of statement **program.** Such files are called Scratch or Temporary files and are automatically deleted when the program is terminated or completed.

We can define file type and variables just like those for arrays/records. There are available standard functions/procedures that operate on files. We propose to discuss all these and the use of files in programs in this chapter.

12.1 File Data Type

A file consists of a sequence of components which must be of uniform type. A file's type is determined by the type of its components. The components form a base of a file. The type of base is always declared. File data type may be defined as:

```
type
  file-type-identifier = file of base-type
```

where *base-type* indicates the type of the components which constitute the file. The file may consist of components of any allowed type, that is, integer, real, char, boolean, enumerated, subrange, record, arrays, and sets except another file type.

Variables of file type can be created by the declaration

```
var
  v1, v2, . . . . . : file-type-identifier
```

where v_1, v_2 . . . . are the variable identifiers of file type. Examples of defining file types and file variables are

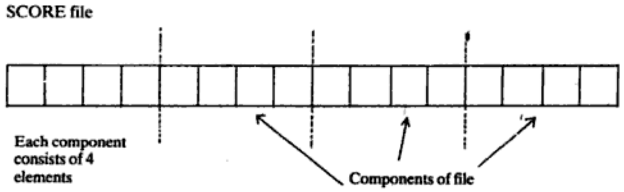
```
type
  NUMBERFILE  = file of real ;
  INTEGERFILE = file of integer ;
  LOGICALFILE = file of boolean ;
var
  ALPHA, BETA : NUMBERFILE ;
  GAMMA       : INTEGERFILE ;
  DELTA       : LOGICALFILE ;
```

here ALPHA, BETA are defined as file variables of type NUMBERFILE and the components of file are all of real type. Similarly, GAMMA is a file variable of type INTEGERFILE and the elements of the file are of type integer. Here DELTA is a file variable of type LOGICALFILE with elements of boolean type.

Components of a file can be arrays and records. This is illustrated by the following example.

```
type
  CLASSFILE              = file of record
     LASTNAME              : packed array [1 .. 10] of char ;
     INITIALS, GRADE       : char ;
     TEST1, TEST2, TEST3, MEAN : real ;
                         end ;
DATAFILE = file of array [1.. 4] of integer ;

var
  SECTION1, SECTION2, SECTION3 : CLASSFILE ;
                         SCORE : DATAFILE ;
```

Here, variables SECTION1, SECTION2, SECTION3, are file variables of type CLASSFILE and their components are records with the fields LASTNAME, INITIALS, GRADE, TEST1, TEST2, TEST3, MEAN. Similarly, SCORE is a file variable of type DATAFILE with the components being integer arrays, each having 4 elements. Schematically we can show the SCORE file as



The number of components in a file can be varied, that is, a file can be of variable length. Components may be added or deleted. End of a file is indicated by the end of file marker. Further examples of file declarations are :

```
type
FLOWERFILE  = file of (ROSE, LILY, NARGIS, DAISY,
                       PANSY, PRIMROSE) ;
VEHICLEFILE = file of (CYCLE, SCOOTER, CAR, BUS, TRUCK)
SETFILE     = file of set of ['A'..'Z'] ;

PARTS = record

          PARTNUM  : integer ;
          PARTDESC : packed array [1 .. 10] of char ;
          PARTSIZE : char
          end ;
PARTFILE = file of PARTS ;
```

File variables of the above type may be created as :

```
var
WHITE, RED     : FLOWERFILE ;        { file of enumerated type
CONVEYANCE     : VEHICLEFILE ;               components }

BOOK           : SETFILE ;     ← file of sets
MOTOPARTS      : PARTFILE ;    ← file of records
```

A file of files is not allowed. Moreover, the specification of files as elements of other structured data types is also implementation dependent. This may be ascertained by reference to the installation Pascal manual.

12.2 File Buffer Variable

Let us indicate the components of a file as comp-1, comp-2, comp-3, ..., comp-n. We can represent a file consisting of these components as

Y↑ ⇒ is a real variable.
P↑ ⇒ is an array of 4 elements : its individual elements are P↑ [1], P↑[2], P↑[3], P↑[4] ; each of the elements is of integer type.
Q↑ ⇒ is a record with fields PARTNUM, PARTDESC, PARTSIZE

The fields of variable Q↑ can be referred to as

Q↑ · PARTNUM {type integer}
Q↑ · PARTDESC [1], Q↑ · PARTDESC [2], ... Q↑ · PARTDESC [10] { type char
Q↑ · PARTSIZE { type char }

Whenenver, we perform a **read/write** operation, we actually manipulate the file buffer variable. Similarly, the buffer variables may be used in expressions and statements exactly in the same manner, as other variables, according to the allowed rules of Pascal language.

12.3 Communication with Files

The buffer indicator always points to a particular component of a file. It can move from one component to the other. This movement is always forward, never in the back direction, except in the case where buffer is to be set to the first component of the file. All communications between the program and the file are done through the file buffer using the following standard functions/ procedures provided by Pascal.

eof { end of file }
rewrite
reset
get
put

These functions/procedures help to read/write and access any component of a file. Their meaning and use are explained below :

(a) *The* **eof** *function*

This is a standard Pascal function and is specified as

eof (*file-identifier*)

Here *file-identifier* indicates the user-defined file name.
An example of use is

eof (F)

where F is the name of file,
The value of this function is true if the buffer pointer has moved beyond the end of file, otherwise, false. Schematically, we can show the action of **eof** (F) as

variable and move the buffer indicator to the next component of the file. It is specified as

get (*file-identifer*)

An example is

get (NUMFILE)

suppose we wish to read the current component of NUMFILE in X and have the buffer pointer advance to the next component, we can do this using the following code :

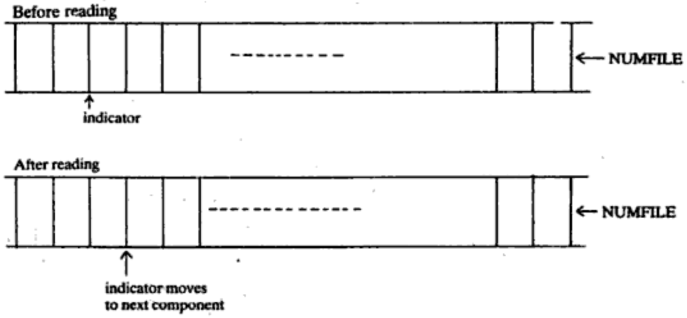
```
var
  NUMFILE : file of integer ;
        X : integer ;

begin
  X := NUMFILE ↑;
  get  (NUMFILE) ; { buffer indicator goes to the next element of file
                                                        NUMFILE}
end ;
```

This is precisely what the **read** statement does. Its action is explained further by the following illustration.

Now, let us suppose, we wish to read component X from file variable named NUMFILE. This is specified as

```
read (NUMFILE, X)
          ↑      ↑
      file-name  component to be read
```

After an element has been read, the indicator moves to the next component of the file. This is illustrated below.



Remember, whenever any data element is to be read from a file, the file must be prepared by the **reset** command. An exception to this rule is the standard file **input.** The system automatically carries out the operation **reset** (input) at the start of the program when **input** is specified as a parameter in the **program** statement. The statement **read** (*file-name, component*) fails if the indicator is already at the end of the file.

(e) *The* **put** *procedure*

The **put** procedure is used to append the current content of the buffer variable to the end of the file. It is specified as

put (*file-identifier*)

An example is

put (NUMFILE)

With this, the following operations get specified:

- the buffer variable NUMFILE ↑ points to an element which gets added as the last element of the file NUMFILE.
- data can be written into this component.

For instance, if specify

NUMFILE ↑ : = X ; **put** (NUMFILE) ;

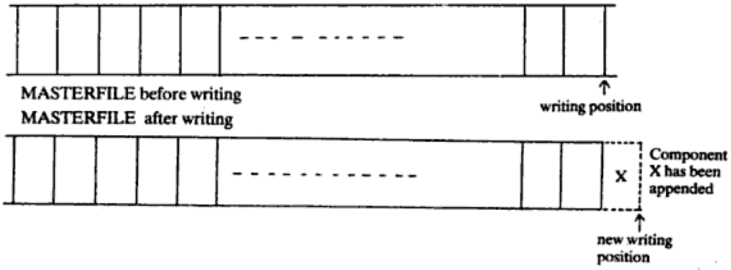
then value of variable X is written as the last element of file NUMFILE.

The above objective may also be achieved with a **write** statement as explained below:

Suppose, we wish to write a component X in a file MASTERFILE. It may be specified as

```
write (MASTERFILE, X)
       ↑           ↑
       file-name   component
```

This statement appends the component X to the file named MASTERFILE and moves the writing position of the indicator to the new end-of-file as indicated below schematically.



The following example illustrates the use of **get** and **put** procedures.

Suppose we read data from **input** file and go on writing on **output** file, till the colon (:) character is encountered. The code for this may be

```
while input ↑ < > ':' do
   begin
     output ↑ : = input ↑ ;
     put (output)
     get (input)
   end ;
```

As another example, consider a file of integer numbers. Nâme it as INTEGERFILE. Compute the product of all the components. See that the product does not exceed **maxint.** When the end of the file is reached, print the answer. A program for this may be developed as follows :

```
program  PRODUCT (INTEGERFILE, output) ;
   var
     INTEGERFILE : file of integer ;
     J, PROD : integer ;

   begin
     reset  (INTEGERFILE) ;
     PROD : = 1;
     while not (eof (INTEGERFILE) )  and maxint >  PROD do

        begin
          read (INTEGERFILE, J) ;
          PROD := PROD * J
        end  ;
     writeln  ('PRODUCT = ', PROD)
   end.
```

The following points may be kept in mind about files:

* File is a structured data type.
* Files may be defined as internal or external to the main memory.
* External files can exist on secondary storage, independent of any program.
* External files can hold much larger volume of data than any other data type.
* All files that exist independently of the program, but are to be used in the program, must be listed as program parameters in the statement **program.**
* Internal files are created and used within the program. Their names are not listed in the program parameters if they are not to communicate with environment. These are the Scratch or Temporary files. They are deleted as soon as the program terminates.

Remember **input** and **output** are predefined files in Pascal. They must never be declared in the program. These are automatically included in the program when specified as parameters with statement **program.**

12.4 Files as Parameters in Subprograms

File variabels can be used as actual arguments of standard and user-defined subprograms. However, when used with subprograms, they must be declared as **var** parameters in the formal argument list. We illustrate this by an example.

Let there be four files, each containing integer components. We wish to compute the sum of all the elements in each file.

* Define four files containing integer elements.
* Introduce a procedure which computes the sum of the elements
* Invoke this procedure to compute the sum for every file.

The program may be as follows:

```
program SUM (AFILE, BFILE, CFILE, DFILE, output) ;
  type
    INTEGERFILE = file of integer ;
  var
    AFILE, BFILE, CFILE, DFILE : INTEGERFILE ;
    procedure COMPONENTSUM (var XFILE : INTEGERFILE) :
       var
         J, SUM : integer ;
       begin
         reset (XFILE) ; { every file must be reset }
         SUM : = 0 ;
           while not eof (XFILE) do
           begin
             read (XFILE, J) ;
             SUM := SUM+J
             end ;
           writeln ('SUM =', SUM)
       end ;
  begin
     { Assume that the files have already integer data elements }
    COMPONENTSUM (AFILE) ;
    COMPONENTSUM (BFILE) ;
    COMPONENTSUM (CFILE) ;
    COMPONENTSUM (DFILE)
  end.
```

Pascal does not allow any operation on the file variables, even assignment. There are no file constants. Moreover, files cannot be used as value parameters even (?).

Example 12.1

Given certain data as:

```
A S G 4 5 6 C A 3
7 K D 5 S G 2 C 9
A
```

Develop a program to do the following:

(i) prepare a master file;
(ii) separate the numeric and non-numeric characters and store them in two separate files;
(iii) sort the non-numeric characters alphabetically,
(iv) obtain the frequency of occurrence of different non-numeric characters

Program 12.1

```
program FILEHANDLING (input, F, CHARFILE, NUMFILE, output) ;
{ Program to separate and store integer numbers and characters, present in
a master file, in two other files NUMFILE and CHARFILE. Then
alphabetize the characters and find the frequency of occurrence of each
character }

type
    PA = array [1 .. 50] of char ;
  ORIGFILE = file of char ;
    CHA = array [1 .. 50] of char ;
  INTFILE = file of integer ;

var
     FILENAME : string { system command }
            F : ORIGFILE ; { stores input records }
     CHARFILE : ORIGFILE ; { stores characters present in master file }
      NUMFILE : ORIGFILE ; { Stores integer numbers present in master
                                file as characters }
         TEMP : char ;
           CH :CHA ;
      A, Z, M : char ;
I, J, T, INTEMP : integer ;
        IFILE : INTFILE ; { stores integers corresponding to integer
                                    characters present in master file }
{------------------- }
procedure SORTLIST (var DATA : CHA ; N : integer) ;
{ Procedure to sort the characters present in a file }
var
  I, J : integer ; { Loop indices }
    MIN : char ;
begin
  for I := 1 to (N-1) do
    begin
      for J := 1 to (N-1) do
        begin
          if (DATA[J] > DATA [J+1] ) then
```

```
            begin
              MIN := DATA [J] ;
              DATA [J] := DATA [J+1] ;
              DATA [J+1] := MIN
            end
          end
        end
      end ; { of procedure SORTLIST }
{------------------------}
procedure FREQUENCY (DATA : CHA ; M : integer) ;
{ Procedure to calculate the frequency }
var
  I, J : integer ; { loop indices }
  FREQ : integer ; { stores frequency of occurrence }
  begin
    for I:= 1 to M do
      begin
        FREQ := 1 ;
        if DATA[I] < > '*' then

          begin
            for J := (I+1) to M do
              begin
                if DATA[J] < > '*' then
                  begin
                    if (DATA [I] = DATA [J]) then
                      begin
                        FREQ := FREQ + 1 ;
                        DATA [J] := '*'
                      end
                  end
              end
          writeln (DATA[I] : 10, FREQ : 19)
        end
    end

end ; { of procedure FREQUENCY }
{---------------}

procedure WRITEFILE (var OUTDATA : ORIGFILE) ;
  { Procedure to write the contents of a file }
var
  I : integer ;
begin
  while not (eof (OUTDATA) ) do
    begin
    TEMP := OUTDATA ↑ ;
```

```
      write (TEMP, ' ');
      get (OUTDATA)
    end ; writeln
  end ; { of procedure WRITEFILE }

  {------------------------------}
  { Main program begins now }
  begin
  { creation of master file }
  FILENAME : = 'MASTERFILE' ;
  assign (F, FILENAME) ; { this command is not a Pascal statement but
                                           system command }
  rewrite (F) ;
  writeln ('Key in the data . . . characters spaced by blank . . .To end, type
                                                                    *') ;
  read (TEMP) ;
  while TEMP < > '*' do
     begin
       F↑ := TEMP ;
       put (F) ;
       read (TEMP)
     end ;
  writeln ;
  reset (F) ;

  {-----------------}
  { Creation of NUMFILE }
       FILENAME : = 'NFILE' ;
       assign (NUMFILE, FILENAME) ; { system command }
       FILENAME : = 'FILEINT' ;
       assign (IFILE, FILENAME) ; { system command }
       J := 0 ;
       writeln ('Give lower and upper limits of the char set') ;
       read (A) ; readln (Z) ;
       rewrite (IFILE) ;
       rewrite (NUMFILE) ;
       while not (eof (F) ) do
       begin
       TEMP : = F↑ ;
       if (ord (TEMP) > = ord ('A')and (ord (TEMP) < = ord ('Z') ) then
         begin
           J := J+1 ;
           CH[J] := TEMP
         end
       else
```

```
         begin
            NUMFILE ↑ := TEMP ;
            put (NUMFILE) ;
            INTEMP := ord (TEMP) ;
            IFILE ↑ := INTEMP ;
            put (IFILE)
        end ;
   get (F)
end ;
     T := J ;
{ Invoke the procedure SORTLIST }
SORTLIST (CH, T) ;
{-----------------------------}
{ Creation of CHARFILE }
  FILENAME : = 'CFILE' ;
  assign (CHARFILE, FILENAME) ;
  rewrite (CHARFILE) ;
  for I := 1 to T do
  begin
     TEMP := CH[I] ;
     CHARFILE ↑ := TEMP ;
     put (CHARFILE)
  end ;
{---------------------------------}
reset (F) ;
reset (NUMFILE) ;
reset (CHARFILE) ;
writeln ('The master file is') ;
WRITEFILE (F) ; writeln ;
writeln ('Digits present in the master file are') ;
WRITEFILE (NUMFILE) ; writeln ;
writeln ('Characters (alphabetized) present in the master file are') ;
WRITEFILE (CHARFILE) ; writeln ;
writeln ('Frequency of occurrence of different characters') ;
writeln ('CHARACTER          FREQUENCY') ;
FREQUENCY (CH, T)
end.
```

Sample input/output

```
Key in the data . . . characters spaced by blank . . . To end, type *
A S G 4 5 6 C A 3 7 K D 5 S G 2 C 9 A *
Give lower and upper limits of the character set
A Z
The master file is
A S G 4 5 6 C A 3 7 K D 5 S G 2 C 9 A *
Digits present in the master file are
4 5 6 3 7 5 2 9
```

Characters (alphabetized) in the master file are

A A A C C D G G K S S

Frequency of occurrence of different characters

CHARACTER	FREQUENCY
A	3
C	2
D	1
G	2
K	1
S	2

12.5 Text Files

A text file consists of lines of characters. Each line is terminated by a end-of-line (e.o.l) marker. Schematically, we can show this as.

$c_1\ c_2\ c_3\ c_4 \ldots\ldots\ldots\ldots c_{n1}$ $\boxed{\text{e.o.l.}}$

$c_1\ c_2\ c_3\ c_4 \ldots\ldots\ldots\ldots c_{n2}$ $\boxed{\text{e.o.l.}}$

$c_1\ c_2\ c_3\ c_4 \ldots\ldots\ldots\ldots c_{n3}$

Here c_1, c_2, c_3, ... represent any Pascal character. The lines constituting the files can be of any length. We see that the text file is a character file. Such a file can be declared as

```
type
  file-type-identifier = file of char
```

In place of this, Pascal allows the declarations of a character file as

```
type
  file-type-identifier = text
```

where **text** is a keyword and indicates 'file of char'. Examples of declaring text files are

```
type
  AUTHORFILE = text ;
  NAMEFILE = text ;
```

File variables of these type may be defined as

```
var
BOOK : AUTHORFILE ;
CLASS : NAMEFILE ;
```

or as

```
    var
      BOOK, CLASS : text ;
```

input and output are the standard text files where lines are manipulated. In fact, these are

```
    var
      input, output ;  text ;
```

These two standard file variables (or simply files) refer to the two standard input and output media of a computer system, say, for example, a terminal, VDU, printer, etc., The file **input** can be examined only so the file procedure **get** is applicable to it. Similarly, the file **output** can be generated only, that is why, file procedure **put** is used with it.

We use **read/write** statements in every program. These, in fact, are the abbreviations introduced for input and output of data :

```
      read (v) ⇒    v= input ↑ ;
                    get (input) ;
      write (e) ⇒   output ↑ : = e ;
                    put (output) ;
```

where *v* is a variable and *e* is an expression. Similarly, **eof** indicates **eof (input)**.

As described above, text files are divided into lines. The statements **writeln, readln, eoln** can be used to control the line-by-line arrangement of **input** and **output** files. More explicitly, the function of these statements is:

writeln ⇒ terminates the current line of the output textfile and writes the end-of-line mark.

readln ⇒ • skips to the start of the next line of the input textile
• the first character of the next line is obtained from **input ↑**

eoln ⇒ gives the value true when the end of the current line of the textfile **input** has been reached.

The reader should understand carefully the above significance of the **writeln, readln** and **eoln** statements which have been used in our programs.

You will observe that text file, though a character file, is distinct from the **file of char**, in the following respects:

- A textfile is divided into lines, each terminated by a special end-of-line marker. These markers are written only with the **writeln** procedure.
- file declared as **file of character** is not divided into lines and there are no end-of-line markers.
- end-of-line marker is a special character which cannot be inserted into the file in the same way as other characters, but is done automatically by the system.
- **read, readln, write** and **writeln** procedures are used with the text files, whereas **get** and **put** procedures are employed with the non-text files.

Exercises 12

12.1. Complete the following sentences:

(a) Pascal supports only .... files.
(b) Pascal programs communicate with the outside environment via ....
(c) External files are always resident on .... devices.
(d) A Pascal file consists of .... whose type must be defined.
(e) The declaration of a file variable automatically introduces a .... called file ....
(f) The value of a file component can be read/written with the help of a ....
(g) The movement of the buffer indicator is always in the .... direction.
(h) The **input** and **output** are .... files in Pascal.
(i) File variables can be used as actual arguments in subprograms, provided they are declared as .... parameters in the .... argument list.
(j) A file is generated by writing components to it using the standard procedures .... and ....
(k) The components of a file can be read by a program using the standard procedures ... . and ....

12.2. Tick the correct answer in the following:

(a) File variables can/cannot be used in assignment statements.
(b) Files can/cannot be specified as value parameters.
(c) A buffer is always/never associated with a file.
(d) During the **read/write** operations, it is the file variable/buffer which is manipulated.
(e) An entire file can/cannot be transferred to a subprogam as a **var** parameter.
(f) A text file is comprised of single/multiple lines of character-type data.
(g) File components may/maynot be of structured type.
(h) The buffer variables have the same/different name as the file.
(i) A file of files is allowed/disallowed.
(j) Data items comprising the components of a file are of the same/different type as the buffer variable.

12.3. Point out mistakes, if any, in the following:

```
type
  DATA = packed file of real ;
  CHAIN = file of array [1 .. 10] char ;
  TELEFILE = file of
                     record
                       NAME : file of char ;
                       ADDR : packed array [1 .. 25] of char ;
                  TELENUM : integer
                     end ;
```

12.4. Indicate completely the buffer variables associated with the following file variables:

```
var
  X : file of integer ;
  Y : file of array [1 .. 5] of char ;
  Z : file of
          record
        REALPART : real ;
        IMAGPART : real
          end ;
```

12.5 State the rules for defining file-type and file-type variable. Give examples.

12.6. Explain the concept of buffer variable. Illustrate your answer by examples and discuss the use of such variables for communication with files.

12.7. Give the standard Pascal functions/procedures that can be used with files. Explain their formats and use.

12.8. What is Text file? How is it defined? Give examples.

12.9. Explain the difference between user-defined text files and standard Pascal text files **input** and **output**.

12.10. Describe the purpose and use of the procedures **readln, writeln** and **eoln.**

12.11. Bring out the differences between

(i) Text file and **file of char**
(ii) **get** and **read** procedures
(iii) **put** and **write** procedures
(iv) Internal file and external file
(v) **eof** and **eoln**

12.12. Explain, in words, the meaning of the following declarations:

(a)
```
var
   OBSERVATIONS : file of real ;
```

(b)
```
var
   NAME, ADDRESS : text ;
```

(c)
```
type
   SAVINGSACCOUNT = 1 .. 20000 ;
   CURRENTACCOUNT = 20001 .. 25000 ;

var
   SAVACTFILE : file of SAVINGSACCOUNT ;
   CURACTFILE : file of CURRENTACCOUNT ;
```

(d)
```
type
   ROWS = array [ 1 .. M ] of integer ;
   MAT = file of ROWS ;

var
   TWODIM : MAT ;
```

Can you say that TWODIM is a matrix? If yes, what is its size?

(e)
```
type
   SIZE      = packed array [1 .. 25] of char ;
   NUM       = 1 .. 99999 ;
   PERSON    = record
                 NAME, ADDR  : SIZE ;
                 TELEPHONE   : NUM
               end ;

var
   PERSONFILE : file of PERSON ;
```

(f)
```
X : array [1 .. 100] of real ;
Y : file of real ;
```

```
(g) begin
      while not eoln (file-name) do
                      get (file-name) ;
      get (file-name)
    end ;
(h) begin
      write (file-name, variable-1, variable-2, . . . .) ;
      write (file-name)
    end ;
```

12.13. Look at the following declarations:

```
type
     CLASSTYPE  = (MA, MSC, MTECH, MPHIL, PHD);

    STDRECTYPE  = record
                    NAME      : packed array [1 .. 10 ]  of char ;
                    CLASS     : CLASSTYPE ;
                    YEAR      : packed array [1 .. 3] of char ;
                    ENROLNO : integer
                    end ;
    STDFILETYPE  = file of STDRECTYPE ;

var
    CLASSX : STDFILETYPE ;
```

Prepare a schematic representation of the complex file CLASSX.

12.14. Assume that a file contains negative and positive integers. Develop a program which does the following (a) counts the number of negative and positive integers, (b) prepares two files, one for each negative and positive integers arranged in descending sequence using the merge-sort algorithm.

12.15. Given two sets of real data obtained from an experiment. Develop a program to do the following:

(a) (i) Prepare two files MFILE and NFILE.
   (ii) Sort the files in ascending sequence and then merge them in ascending order into a file PFILE.
   (iii) Print the output as ;

```
MFILE : . . . . . . . . . . . .
NFILE  : . . . . . . . . . . .
PFILE  : . . . . . . . . . . .
```

(b) Next, prepare another program that reads data in arrays M and N. Arrange the data in ascending sequence and merge the arrays into another ordered array P. Prepare the output as

```
MARRAY : . . . . . . . . . .
NARRAY : . . . . . . . . . .
PARRAY : . . . . . . . . . .
```

Compare and contrast the program designed using files and arrays.

12.16. Given a book record of the following type:

```
type
   BOOK = record

            TITLE  : packed array [1 .. 100] of char ;
```

12.18. Prepare a Pascal file for the following text :

Sometimes, people argue that hardwork and honesty do not always pay. This is wrong. It is only honesty and hardwork that bring pleasures into life. Can we evaluate pleasures?

Develp a program to transfer this text to another file line-by-line.

Chapter 13

Set Operations and Data Type

We have studied the structured data type arrays, records, and files. They consist of a group of components. Each component can be accessed and manipulated, but the entire array, record or file cannot be processed by a single operation. Operators, which operate on the complete arrays, records or files, are not defined in Pascal. However, Pascal defines another data type—called Set—which is a structured data type and consists of a collection of components. The set can be manipulated either as a complete unit or as a subunit. The subunit may consist of one or more elements. Such data are frequently used in mathematical applications involving sets or in situations where there may be several possible events, and a track of exactly which event takes place, or otherwise, has to be maintained. Moreover, sometimes development of programs may be more logical and appropriate using sets than otherwise. We shall study the data type sets and their use in the following sections.

13.1 Set definition and Elementary Operations

A set is defined as a collection of elements or items, all of the same type. The elements which constitute the set are called its Members. Sets are generally represented by writing its members within braces { }, but we shall use the square brackets, [ ], as is the notation in Pascal. The set of square brackets is referred to as Set Constructor.

Suppose A, B, C, D, E, F is a collection of elements. Then

[A, B, C, D, E, F}

is a set whose members are A, B, C, D, E, F.
Similarly if 1, 3, 5, 7, 9, 11, 13 is a group of odd integers, then

[1, 3, 5, 7, 9, 11, 13]

is a set of odd integers consisting of 7 members. In other words, set elements are 1, 3, 5, 7, 9, 11, 13.
A set may consist of any number of members, but all of the members must *be of the same type.* Other examples of sets are

[A, I, O, U, E]	set of vowels
[RED, BLUE, GREEN, BLACK]	set of colours
[EARTH, MARS, NEPTUNE, MOON, MERCURY, SATURN, JUPITER]	set of planets

[FATHER, MOTHER, SON, DAUGHTER] set of family members
[SYNERR, EXECERR, EXPOVERFLOW, EXPUNDERFLOW, DEVICERR] set of errors

and so on. Some important characteristics of sets and operations allowed on them are:

- the number of members in a set defines its size. Thus, the size of set [A, B, C, D] is 4,
- the order in which the set members are listed is immaterial. For example, the set

[A, B, C, D, E, F]

may be written as

[A, B, E, C, F, D]

or as

[D, E, A, B, F, C]

or a variation of this. All arrangements refer to the same set.

- repetition of some members does not affect the size of the set. For instance, the set

[A, B, A, C, C]

is the same as

[A, B, C]

- a set with no member is referred to as an Empty set and is simply specified as [ ].

Let us define three sets X, Y and Z as:

X = [A, B, C, D], Y = [A, B, C, D]
Z = [A, C, D, E, F]

The following operations are defined on sets

- *Set equality*
  Two sets are equal if they have the same number and type of elements. Thus, sets X and Y are equal.

- *Set union*
  Two sets may be joined. The resultant union is a set of all elements which are members of the individual sets. For example, the union of sets X and Z is the set

  [A, B, C, D, E, F]

  while the union of sets X and Y is [A, B, C, D], that is, same as X or Y.

- *Set difference*

  The difference of two sets is defined as a set which consists of elements which are not common to either set. For instance, the difference set for the sets X and Z is

  [B, E, F]

- *Set intersection*

  The intersection of two sets is defined as a set which consists of all elements common to both the sets. For example, the intersection of set X and Z is the set

  [A, C, D]

- *Set inclusion*

  A set P is said to be included in set Q, if all items in P are also in Q, but not necessarily vice versa. For instance, consider

  P = [1, 2, 3]
  Q = [1, 2, 3, 4, 5]

Then set P is included in set Q. Set P is referred to as subset of Q.

Remember: an empty set is included in every set. Moreover, an element is either a member of a given set or is not.

13.2 Set Constants

Members of a set are said to constitute its Base. All the members must be of the same type. Set constants are defined in Pascal by enclosing the members between square brackets and separating the members by comma. Examples of set constants are:

['A', 'B', 'C', 'D', 'E', 'F']

[4, 6, 8, 10, 14, 16]

[TRUE, FALSE]

[APR, MAY, JUN, JUL, AUG]

Set constants can also be defined using arithmetic integer expressions as

[K+1, J+2, J*L]

assuming that K, J, L are integer identifers which have been defined earlier.

The Base type of a set constant must be an ordinal data type, that is, integer, boolean, char, enumerated and subrange type. This restriction allows an efficient implementation of Pascal.

An example of set constant using subrange type is

[1 .. 13, 23, 29]

The members of the set constant are 1, 2, 3, .. 13, 23, 29.

Pascal does not allow a set to be declared in the **const** declaration.

Here RAINBOW is a set variable and its base consists of VOILET, INDIGO, ... RED as its members. Another example is

```
var
   RANK : set of (FIRST, SECOND, THIRD) ;
```

The set variable RANK can assume any of the following set values

```
[FIRST, SECOND, THIRD]
[FIRST, SECOND]
[SECOND, THIRD]
[THIRD, FIRST]
[FIRST]
[SECOND]
[THIRD]
[ ]
```

Standard Pascal does not put any limit on the number of members a set may have, however, restriction may be imposed due to the computer hardware. Many compilers limit the number of members of a set to 256.

When defining a set **type** with integer and character as *base-type*, then base-type must be a continuous subrange of integers or characters. Commas are not allowed to be used with these. For example, the following type declarations are not permitted:

```
type
   LETTERSET = set of 'A', 'B', 'C', 'D', 'E' ;
   DIGITSET = set of 1, 2, 3, 4, 5, 6, 7 ;
```

The correct **type** declaration is

```
type
   LETTERSET = set of 'A' . . 'E' ;
   DIGITSET = set of 1 .. 7 ;
```

Comma can only be used with the set declaration of enumeration base type. The following declaration is allowed:

```
type
   OPSET = set of (+, −, /, *) ;
   VOWELS = set of (A, E, I, O, U) ;
```

The reader should note that A, E, I, O, U are not enclosed with quotes. They will not be treated as characters but simply as symbols.

Assignment statement can be used to assign a set constant to a set variable. Examples are

```
WORKINGDAYS    := [MON, TUES, WED, THRS, FRI] ;
  PRIMENUMBERS := [1, 3, 5, 7, 11, 13, 17, 19, 23] ;
  COMPUTERSIZE := [MICRO, MINI, MAINFRAME, SUPER] ;
  LANGUAGES    := [APL, PASCAL, FORTRAN, BASIC, COBOL,
                               LISP, PROLOG, ADA] ;
```

13.4 Set Operators and Expressions

Set constants and variables can be combined to design set expressions by the use of set operators. The following operators are available in Pascal which can be used with sets.

Operator	*Set function*
+	Union
−	difference
*	intersection
=	equality
< >	inequality
<=, >=	inclusion
in	membership

In order to use these operators with sets, it is necessary that two set types are compatible.

Two set types are said to be compatible when the following conditions are satisfied:

- both have the same base type,
- the base types are subranges of the same type,
- one of the base types is a subrange of the other.

Following examples illustrate the use of set operators:

We define two compatible set variables S1 and S2 as

S1 = ['A', 'B', 'C', 'D' 'E'] ;
S2 = ['A', 'I', 'O', 'U', 'E'] ;

The union operator, +, joins the two sets and gives a new set S (say) as

S : = S1+S2 ⇒ ['A' 'B', 'C', 'D', 'I', 'O', 'U']

The set difference operator, −, yields the set X

X = S1 − S2 ⇒ ['B', 'C', 'D', 'I', 'O', 'U']

With the set intersection operator, *, the result of

S1*S2

is a set ['A', 'E']
The set equality operator, =, when used with sets S1 and S2 as

S1 = S2

yields the result as false, because sets S1 and S2 are not identical.

With the set inequality operator, the expression

S1 < > S2

yields the result as true because sets S1 and S2 are unequal.
With the set inclusion operator, <=, if we specify

```
set X <= set Y
```

this implies that set X is a subset of set Y ; while if we use

```
set X >= set Y
```

then it indicates that set Y is a subset of set X. Hence,

```
S1 <= S2   false
S1 >= S2   false
```

The operator **in** enables to determine whether a given item is a member of a set. The result of this operation is boolean type. The operator **in** is used as

---

item-x **in** *set-x*

---

where *item-x* is the element to be searched in *set-x*. If the *item-x* is present in *set-x*, the result of **in** operation is true, otherwise false.

For example,

'A' **in** S1 ⇒ true

whereas

'P' **in** S1 ⇒ false

The left operand of operator **in** must be an ordinal type and its right operand must be of a set type having elements of that (same) ordinal type.

The reader should note that the operators +, −, * yield the result as another set, while operators =, < >, <=, >= and **in** lead to boolean constants as their result.

Suppose a student is eligible to be admitted to a course only if his age is between 17 and 25 years and he has scored marks in the reange 60-100. This condition can be expressed as

```
(17 >= AGE) and (AGE <= 25) and (60 >= MARKS)
                    and (MARKS < = 100)
```

This can be expressed more easily by using sets as

```
(AGE in [17 .. 25] ) and (MARKS in [60 .. 100] )
```

Any subrange, whose beginning value is larger than its ending value, is considered to contain no elements, so that

```
[  ] = 'Z' .. 'A'
```

is true

Use of sets helps to write programs in a simpler way. To illustrate this, we consider the following example

Program 13.1

```
program HEXADECI (output) ;
   { Program to convert a hexa number to decimal form }
   var
      HEXCH, NO : char ;
      DECINUM, SUM : integer ;
      DIGSET, LETSET : set of char ;
   begin
      SUM : = 0 ;
      DIGSET : = ['0' .. '9'] ;
      LETSET : = ['A' .. 'F'] ;
      { Read a character }
   repeat
      write  ('Enter the hexa digit? . . .') ;
      readln  (HEXCH) ;
         if (HEXCH in DIGSET) or (HEXCH in LETSET) then
         begin
            if HEXCH in DIGSET then
               DECINUM : = ord (HEXCH)−ord ('0')
            else
               DECINUM : = (ord (HEXCH)−ord ('A')) + 10 ;
               SUM : = SUM * 16 + DECINUM
         end ;
      writeln ('Decimal equivalent of hexa digit =', DECINUM) ;
      write ('Do you want to enter the next hexa digit? (Y/N) ..') ;
      readln (NO)
   until (NO = 'N') ;
      writeln ;
      writeln  ('Decimal equivalent of the complete hexa number =', SUM) ;
      writeln ('Stop')
   end.
```

Sample input/output

```
Enter the hexa digit? . . . 3
Decimal equivalent of the hexa digit = 3
Do you want to enter the next hexa digit? (Y/N) .. Y
Enter the hexa digit? . . . A
Decimal equivalent of the hexa digit = 10
Do you want to enter the next hexa digit? (Y/N) .. Y
Enter the hexa digit? . . . 6
Decimal equivalent of the hexa digit = 6
Do you want to enter the next hexa digit) (Y/N) .. N
Decimal equivalent of the complete hexa number = 934
Stop
```

Sample input/output

```
Enter the values of J, K, L, M
5  11  30  3
Number of elements in the set .... 25
Enter the values of J, K, L, M
5  14  50  2
Number of elements in the set .... 44
Enter the values of J, K, L, M
0 −1  0  0
Number of elements in the set .... 0
Stop
```

13.5 Set Operator Hierarchy

A set expression may be designed using set constants, variables and operators. When several opeators appear in an expression, priority of the set operators is as :

Intersection	*
Union	+
Difference	−
Membership	**in**

Expressions are evaluated from left to right as usual.

Let us illustrate these rules further.

Let set

X = [1, 3, 5, 7, 9, 11, 13]
Y = [2, 4, 6, 8, 10, 12]
Z = [3, 4, 5, 7]

The expression

X + Y * Z

is evaluated as:

- first compute intersection of sets Y and Z, that is find Y * Z; say, the result is set W ;
- next find the union of set X and set W ;
- the result is the set which is the value of the given set expression X + Y * Z

The student is urged to write the value of the set expression X + Y * Z.

Many useful programs may be designed using sets. An important application is in data processing. It is commonly required to examine whether the data being input to a program is correct and lies in the given range. This is checking the validity of the input data.

The following program code goes on reading character data and storing in array CHARRAY till any of the character '*', '/', ')', '=', '(' is encountered in the data.

```
const
  N : 1000 ;

var
  J : integer ;
  CH : char ;
  CHARRAY : packed array [1 .. N] of char ;

begin
  J : = 1 ;
  repeat
    readln (CH) ;
      while CH in ['A' .. 'Z', '.', '?', ',' ';'] do
        begin
          CHARRAY [J] : = CH ;
            J : = J+1
        end
until CH in ['*', '/', ')', '=', '(']
end.
```

The set type data structure provides a convenient way to carry out several kinds of checks, such as membership testing, error checking, simplifying boolean expressions designed using several **or** operators, and so on.

Exercises 13

13.1. Tick the correct answers.

(i) Set is a simple/structured data type.
(ii) The base of a set may consist similar/disimilar types.
(iii) Set constants can/cannot be defined using arithmatic integer expressions.
(iv) The operator = implies equality/assignment of values.
(v) The **in** operator needs one/two operands.
(vi) The value returned by the operators +, −, * is boolean/set.
(vii) Value of a set can/cannot be read by **read** statements.
(viii) Set expressions can be designed using set/arithmetic/both constants.
(ix) Built-in functions are/not defined for sets.
(x) Record fields can/cannot consist of sets.

13.2. Complete the following sentences

(a) A set is defined as a collection of .... called .... type.
(b) The square brackets [ ] are called ....
(c) Joining of two sets is referred to as ....
(d) Set may be constructed using .... statemetns.
(e) The number of elements in the base of the set VEHICLE : = [CAR, CYCLE, SCOOTER, BUS, VAN, TRUCK, AEROPLANE, BULLUCKCART, TONGA] is ....
(f) The allowed set operators are ....
(g) A null set consists of ....
(h) Set type is defined as ....
(i) Two sets type are said to be compatible when ....
(j) Logical operators may be used with .... expressions.

13.3. Define a set. Give examples and determine the size of each set.
13.4. Explain the various operations which are allowed on/between sets.
13.5. How is a set constant defined? Illustate by examples.
13.6. What do you understand by the base of a set? Given the base types permitted in Pascal.
13.7. Define set Type and Variables. What rules must be observed while defining the base-type?
13.8. Let P, Q, R be set variables which have been assigned the following values :

```
P := [1 .. 5, 10 .. 5] ;
Q := [5 .. 10] ;
R := [1 .. 10, 13 .. 15] ;
```

Evaluate each of the following expressions :

(i) P+Q*R
(ii) P*Q*R
(iii) (P−Q) * (Q−R)
(iv) (P+Q) * R
(v) (8 **in** Q) **or** (13 **in** P)
(vi) P <= (Q+R)
(vii) 11 **in** Q * R

13.9. Below are given correct and incorrect set expressions. Separate them and correct the incorrect expressions.

(i) 'A' **not in** 'A' ... 'H'
(ii) **not** 'A' **in** 'A' ... 'H'
(iii) [−5 .. +3] * [+5 .. 10]
(iv) [MON, TUES, WED] + [A, E, I, O, U]
(v) [1 .. 4] = [1, 2, 3, 4]
(vi) [A] **in** ['A' .. 'Z']

13.10. What is the difference between the following :

```
(a)   type
        X = set of 1 .. 13 ;
        Y = 1 .. 13 ;

(b)   type
        A = set of '0' .. '9' ;
        B = set of 0 .. 9 ;
```

13.14. Prepare a statement using sets for the following :
You can join army only if you are between the age of 21 to 28, your height is between 165-190 cm and weight lies in the range 50-70 Kg.

13.12. (a) Suppose a variable NUMBERSYSTEMS is defined as :

```
var
  NUMBERSYSTEMS : set of [BINARY, OCTAL, DECIMAL, HEXA] ;
```

How many possible values can this variable assume?

(b) Give the total number of members of the set H = [5 .. 13, 17, 29 .. 40].

13.13. Data consisting of characters, digits and other special symbols are being entered from the keyboard. We want to count the number of (i) vowels (ii) consonants, (iii) digits and (iv) special symbols in the incoming stream of characters. Develop a program which does this.

13.14. Prepare a file of Pascal reserved words, standard functions, operators and character set symbols. Develop a program which prints all the reserved words and standard functions.

13.15. Develop an algorithm and a program to prepare the concordance (a list of words in alphabetical order and their frequency) of all user-defined identifiers in a given program.

13.16. Define variables of record type, file type and set type. Try to read and write values for such variables. List the error messages obtained on your system.

13.17. How many members may be in a set on your computer system? Does it allow a declaration of the kind **set of char?**

15.18. Develop a program, using sets, that will take a sequence of integers and find the presence, or otherwise, of duplicates. Display the duplicates.

13.19. Suppose there are data consisting of numbers and characters . , ( ) ? * /. You are required to compute the sum of numbers only. Develop a program to do this. Your program should point out and print when any of these symbols is detected. (Think of applications of such programs).

13.20. Design an algorithm and a program (using procedure subprogram) that reads a letter and decides whether it is an upper case or lower case letter. If it is a lower case letter, change it into a upper case letter.

13.21. Prepare a general Pascal program that reads characters from keyboard and counts the number of consonants, vowels and integer digits. Make use of set data structure.

13.22. Develop an algorithm and a program which finds the frequency of different letters contained in a given word of arbitrary length. Generate a table which lists the results as :

```
THE WORD IS

LETTER                    FREQUENCY
...............           ...............
...............           ...............
...............           ...............
...............           ...............
...............           ...............
...............           ...............
```

13.23. Devise a program for Exercise 7.11 using sets.

Chapter 14

Pointers and Dynamic Data Structures

You are aware that memory is allocated to variables in a program block at the time when the block execution is to begin. Moreover, it remains in existence as long as the block is executing. Such variables are referred to as Static Variables. Static variables are referenced by user-defined names. Storage assigned for such variabels cannot be altered during the execution of program. This has the disadvantage that if an excess memory has been assigned in the beginning, some of the memory will remain unutilized, while if the memory assigned happens to be less, the program will have to be recompiled after deciding the appropriate size of the memory.

In order to avoid either of these conditions, and have better control on memory allocation, we define what are called Dynamic variables. They may be simple or structured. With dynamic variables, memory can be allotted or released during program execution. Such variables are not introduced by user-defined names but by Pointers. In Pascal, pointer is a data type with which pointer variables (also called dynamic variables) can be declared. We shall study pointer data types and their use to define new data structures.

14.1 Pointer Data Type

Such data type are declared as

pointer-type-identifier = ↑ *base-type*

The pointer type is defined by prefixing the up-arrow (↑) with the *base-type*. The *base-type* defines the type of data items. A pointer variable, that is declared to be of a certain type, can only point to data items of the type specified by the *base-type*.

Once a pointer type has been defined, we can use the **var** declaration to declare pointer variables :

var
  v_1, v_2, ... : *pointer-type-identifier*

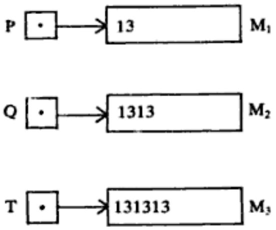
Consider the example

```
type
  POINTER = ↑ real ;
var
   A, B : POINTER ;
```

Here variables A and B have been defined as pointer variables of type POINTER. The type of data items, to which the pointer points, is real.
Further examples are

```
type
  LINKPOINTER = ↑ integer ;

var
  P, Q, T : LINKPOINTER ;
```

Here P, Q, T have been defined as pointer variables, or simply pointers, of type LINKPOINTER. These pointer variables contain the address of the memory locations which store integer data. Here *base-type* is **integer.** Schematically, we can show this as



Here M_1, M_2, M_3, are memory locations and can store data of integer type. P, Q, T are the pointers (or pointer variables) which store the addresses of locations M_1, M_2, M_3.

In other words, these pointers point to locations M_1, M_2, M_3. Thus, the value associated with the pointer variables is not the content of the memory locations, to which it points, but the address of that location. In other words, pointer is an address of some memory location.

Data values are stored in locations to which the pointer variables point. To refer to this information, the pointer variable name, followed by an up-arrow (↑) is specified. It is called a Reference Variable or Associated Variable.

Pointer variable ↑ ⇒ Reference or associated variable

Thus, the associated variables with the pointers P, Q, T are P↑, Q↑, T↑. These are the Reference variables.

The values stored in locations, to which the pointer points or refers to, can be assigned to the associated variables. For instance, if 13, 1313, 131313 are the values stored in the locations M_1, M_2, M_3, then

```
P↑ := 13 ;
Q ↑ := 1313 ;
T ↑ := 131313 ;
```

The reference variables, such as P↑, Q↑, T↑, act like the regular variable

DATE has three fields (scalar) DAY, MONTH and YEAR. These fields may be accessed as

```
Z↑.DAY    := 13;
Z↑.MONTH  := 4;
Z↑.YEAR   := 1986;
```

But, it is illegal to specify

```
Z.DAY := 13;
```

because Z is a pointer variable and not a regular variable.

Refer to type RPTR. Here two types called RPTR and DATE have been introduced. DATE is of record type while RPTR is of pointer type as indicated by the uparrow. It is important to note that the type DATE occurs with RPTR before it is defined. *This is referred to as Forward reference. The specification with pointers is one of the few places where such a reference is allowed in Pascal.*

Another example of forward reference is

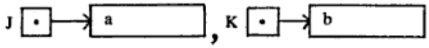
```
type
  NEXTNODE = ↑ NODE;
        NODE = record
              NUMBER = integer;
                  NEXT = NEXTNODE
                     end;
```

Now two types NEXTNODE (pointer type) and NODE have been defined together and afterwards NODE is defined as of record type. Moreover, NEXT is of type NEXTNODE. Such data types are useful to define linked lists and other data structures (Section 14.5)

Every pointer type includes **nil** amongst its possible values. This is a default option. Moreover, **nil** points to no element at all.

14.2 Operations on Pointers

Let J and K be pointer variables, and suppose the values stored in the locations, to which they point, are a and b :

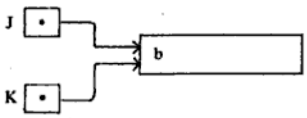


The following operations may be performed on the pointers J and K..

⇒ **Assignment**

```
J := K        (I)
```

this implies that J and K point to the same location and may be indicated as :



The location to which J was pointing is 'freed' and is no more accessible to the program.

A pointer can be assigned the value **nil** as

K := **nil** ;

when the pointer points nowhere. Remember **nil** is a Pascal reserved word.

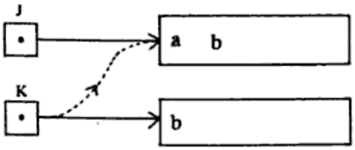
The values stored in locations pointed at by J and K can be assigned to the associated variables as :

```
J↑ := a ;
K↑ := b ;
```

If we specify

```
J↑ := K↑          (II)
```

then the value a is replaced by the value b.
Pictorially, this implies



The reader should note carefully the difference between the assignment (I) and (II). In (I), the address is assigned to J while in (II), value is assigned to J↑.

It is illegal to specify

```
J↑ := K ;
```

or

```
J := K↑ ;
```

because types mismatch. J and K are pointers (or pointer variables) while J↑ and K↑ are associated or reference variables.

Remember, a pointer value may be assigned to another pointer of the same type, while **nil** may be assigned to any pointer.

⇒ Comparison

Only two comparison operators = and < > are allowed to be used with pointers.

The expression

```
J = K
```

compares the addresses as specified by the pointers J and K while

```
J < > K
```

Suppose, we wish to create an unnamed real variable and store the pointer to it in (pointer) variable X. We can do so as

```
new (X)
```

Thus, the new created reference variable is X↑. We can assign a real value to variable X↑. In fact, X↑ can be used at every place where real variables can be employed.

If we specify

```
new (Y)
```

We create a new record variable Y↑ of type DATEPOINTER. The fields of variable Y↑ can be accessed as

```
Y↑.DAY,  Y↑.MONTH,  Y↑.YEAR
```

Next, suppose we wish to destroy the dynamic variables X ↑ and Y ↑. We need to specify

```
dispose (X) ;
dispose (Y) ;
```

Remember, the arguments of the procedures **new** and **dispose** must be *pointer variables and not reference variables.*

The reader should keep in mind that memory space for the reference variable is allocated only after the specification of the procedure **new**. *Similarly, memory space is deallocated after the use of* **dispose** *procedure.*

The pointer variables and the above two procedures find extensive use with data structures such as lists, trees and stacks. Such data strucutres may be more efficient in certain applications as compared to the standard ones. As for example, linked lists are more efficient as compared to the sequential lists where frequent insertions and deletions are made. However, linked lists need more memory space than the corresponding sequential list.

The actual effect of **dispose** procedure may depend on the Pascal implementation which may use **dispose** to destroy storage or just retrieve it.

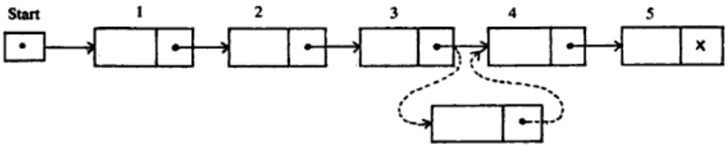
The pointer data types offer the following conveniences :

—the storage size can be adjusted dynamicaly as desired by the program,
—storage may be shared among several variables,
—complex data structures may be designed, updated and manipulated selectively

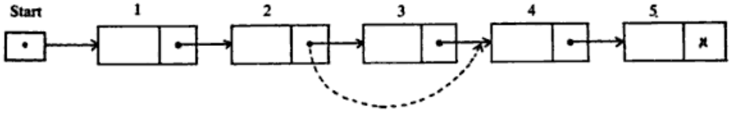
while the possible drawbacks are

—an attempt to update/reference a location, which has already been disposed of, might create problems as the storage may have been allocated for some other use,
—an attempt to refer to the contents of a storage location to which a pointer has not been set up, causes an execution error.

Insertion and deletion of elements in a linked list is illustrated by the following diagrams :



Node inserted between nodes 3 and 4

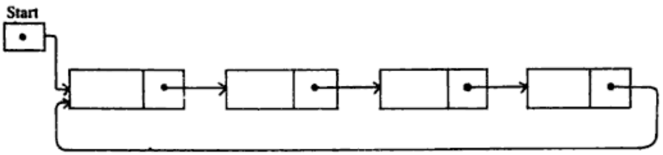


Node 3 deleted

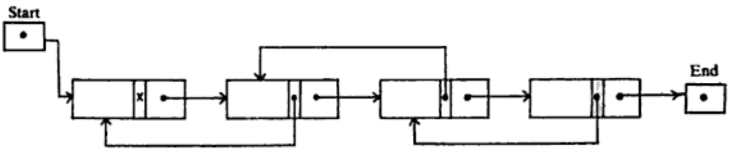
The insertions and deletions of nodes may be anywhere in the linked list.

The components of the linked list considered above are linked together in a sequential manner. This is the simplest type of linked list. Other kinds of lists can also be defined. Examples are circular lists, doubly linked lists, doubly linked circular lists and so on.

A circular list (ring structure) is a linear list having no start and no end. It may look as



In a doubly linked list, there are two pointers associated with each node - a forward ponter and a backward pointer. The structure of such a list may appear as :

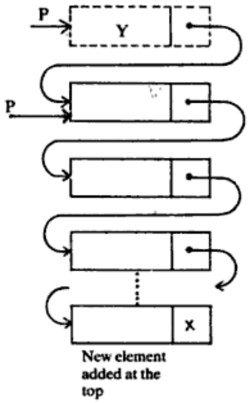


Here cross x indicates **nil**. The double set of pointers enables us to traverse the list in either direction, that is, from start to the end or vice versa.

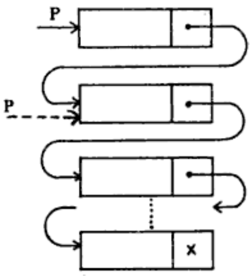
There are available several other kinds of data structures in computer science, such as stackes, queues, trees and so on.

(b) *Stack*

A stack is a linear structure in which items may be added or removed only at one end. We show diagramatically a stack as (where P is a stack pointer)



New element added at the top



Top element deleted

The last item added to a stack is the first item to be removed. Due to this stacks are also called Last-In First-Out (LIFO) list.

Let us define the nodes of the stack as

```
type
  STKPOINTER = ↑ STKNODE ;
     STKNODE = record
     INFMORM : type-identifier ;
        NEXT : STKPOINTER
               end ;
var
  P, T : STKPOINTER ;
```

An element with information Y may be loaded on the stack with the following statements :

```
new (T) ;
   T↑.INFORM := Y ;
      T↑.NEXT := P ;
            P := T ;
```

Similarly, we can remove the top element of the stack and put the data into Y as :

```
    if P = nil then writeln('stack is empty')
      else begin
              Y := P↑ ·INFORM ;
              P := S↑ ·NEXT
           end ;
```

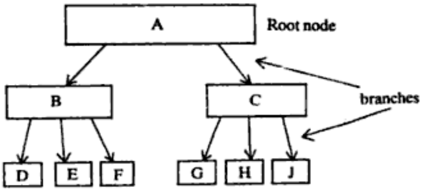
These two stack operation are shown in the above stack figures.

(c) *Queue*

A queue is a linear list of items in which items can be added only at one end (rear) while items can be removed only at the other end (front). As the first item in a queue will be the first item to be moved out of the queue, so queues are also called first-in-first out (FIFO) lists. (Compare queue with a stack).

(d) *Tree*

A tree is a data structure which consists of nodes and branches. These are organized in such a way that they represent some structuring of data. An example of a tree is given below.



When every node has two branches, it is called a Binary tree.

Data structure arrays, lists, stacks and queues are also characterized as of linear type, whereas the data structure tree is a nonlinear data structure.

You have seen that anonymous memory locations may be created/destroyed at anytime during program execution by pointers. Elements may be added/removed to a data structure as and when required at run time. Such structures are said to be Dynamic data structures. Examples of dynamic data structures are: lists, stacks, queues, trees, graphs, etc. They differ from static data structures (arrays, records, files, sets) in that the number of components can be altered and even the relationship among them may be adjusted. Moreover, static data structures are defined by the language while the dynamic data structures may be defined, designed and implemented by the user via pointer data types.

Here, we have described simple type of dynamic data structures. A variety of data structures can be designed using these structures or others. Design of a data structure depends on how the information is to be organized. Data structures are very powerful means of information representation.

Pointer data types help us to implement the various kinds of data structures. We shall not go into these details as these are covered in separate books on data structures.

We shall illustrate the use of pointers and linked list by a program. You are already familiar with searching algorithm by the method of exchanges. This can also be implemented via pointers and linked lists. The following program example illustrates this.

Example 14.1

Design a program, using pointers and linked list, to sort a given list of numbers in ascending sequence using the method of exchanges.

program 14.1.

```
program EXCHSORT (input, output) ;
type
        POINTER = ↑NODERECORD ;
   NODERECORD = ↑record
                  DATA : integer ;
                  LINK : POINTER ;
                  end ;
     DATAARRAY= array [1 .. 20] of integer ;

var
   ARRAYOFDATA : DATAARRAY ;
   DATAPOINTS : integer ;          { Stores the no. of points to be linked }
     I, J : integer ;                              { Loop control variables }

P, P1, K, PTR : POINTER ;
LIST : POINTER ;                               { External points to the list }
BACK : POINTER ; { Keeps track of the node prior to one being examined }
CURRENT : POINTER ;          { Pointer pointing to node being examined }

begin { main program body starts }
   writeln ('Enter the number of data points') ;
   read (DATAPOINTS) ;
   writeln ('The unsorted list is') ;
   for I := 1 to DATAPOINTS do
   read (DATAARRAY [I] ) ;
   writeln ;
{ Create the linked list }
   new (LIST) ;
LIST ↑. DATA := (DATAPOINT–J) do
   new (PTR) ;
PTR ↑. DATA := DATAARRAY [2] ;
   LIST ↑. LINK := PTR ;
   CURRENT := PTR ;
for I := 3 to DATAPOINTS do
```

```
    begin
      new (PTR) ;
      PTR ↑. DATA : = DATAARRAY [I] ;
      CURRENT ↑. LINK : = PTR ;
      CURRENT : = PTR ;
      end ; { for loop for creating list }
      CURRENT ↑. LINK : =nil ;
    for J  : = 1 to (DATAPOINTS−1)  do
      begin
        for  I=1 to (DATAPOINTS−J)  do
      begin
        BACK : = LIST ;
        P1 : = LIST ↑. LINK ;
      while P1  < >  nil do
        begin
          if P1 ↑. DATA  <  BACK ↑. DATA
          then
            begin  { interchange the points in the two nodes }
              new  (K) ;
              K ↑. DATA : = P1 ↑. DATA ;
              P1 ↑. DATA : = BACK ↑. DATA ;
              BACK ↑. DATA : = K ↑. DATA ;
              dispose  (K)
            end ;  { of interchange }
          P1 : = P1 ↑. LINK ;
          BACK ↑ : = BACK ↑. LINK
        end    { while }.
      end    { inner for }
      end ; { outer for }
      writeln  ('The sorted list is').;
        P : = LIST ;
        while (P  < >  nil)  do
          begin
            writeln  (P ↑. DATA) ;
            P : = P ↑. LINK
            end    { print list }
   end. { main }
```

Sample input/output

```
Enter the number of data points
8
The unsorted list is

12   56   90   0   45   76   81   92

The sorted list is

0   12   45   56   76   81   90   92
```

Compare the program of Example 14.1 with the simple sort-exchange program given in Program 9.3.

The reader should appreciate that programs developed using pointers and lists are generally lengthy and need more memory space. However, they offer the advantage of flexibility, creating new and relevant (to the problem) data structures.

Pointer data types have been often used to implement and design various data structures and solve problems in varied areas such as recursion, sparse matrices, polynomial arithmetic, error-correcting codes, systems programming, and so on. Dynamic data structures enable better memory management and efficient access to memory space.

Exercises 14

14.1. Complete the following sentences

(a) Memory is allocated to variables in a program block at the time when the block .... is to begin.
(b) Storage assigned to .... variable cannot be altered during program execution.
(c) Memory can be allotted or released during .... execution with .... variables.
(d) .... data type is used to define dynamic variables.
(e) Variables associated with the pointers are also called .... variables.
(f) Pointers can be associated with both .... and .... data types.
(g) Every pointer type includes .... amongst its possible values.
(h) A pointer value may be assigned to another .... of the .... type.
(i) The comparison operators allowed with pointers are .... and ....
(j) Pointer variable can be .... and .... as and when desired.

14.2. Tick the correct answers

(a) Dynamic variables help to have better/poorer control over computer main memory allocation.
(b) Forward/backward reference is allowed with pointer specifications.
(c) Null pointer points to no/last element of a list.
(d) Pointer variable can/cannot be passed as a parameter in procedure arguments.
(e) A function can/cannot have pointer as its result.
(f) A pointer always points to an anonymous/known variable.
(g) Static variables are referenced by names/pointers.

14.3. What is a pointer? How is it defined? Illustrate by two examples.

14.4. Explain the way pointer and referenced variables are related to each other.

14.5. Illustrate, by examples, the association of pointers to scalar and structured data types.

14.6. What do you understand by Forward reference in Pascal? Give examples.

14.7. Describe the various operations that can be performed on the pointer variables. Give two examples for each.

14.8. Bring out the differences between static and dynamic variables. Explain their advatnages and disadvantages.

14.9. Which Pascal procedures are used to create and annihilate (destroy) pointer variables? Explain their formats and give examples.

14.10. Discuss the possible advatnages and drawbacks of having pointer data types in a language.

14.11. Design a procedure which makes pointer P to point to the same location to which Q points, and releases the memory location to which P pointed previously.

14.12. Explain the following :

(a) List (b) Stack (c) Queue (d) Tree

14.13. Consider the following declarations :

```
type
  SIZE = (SMALL, MEDIUM, LARGE) ;
  POINTER = ↑ SHIRT ;
                  SHIRT = record
                    TSHIRT : SIZE ;
                    NEXTSIZE : POINTER
                      end ;
var
  MEN, BOYS : POINTER ;
```

List the following :

(a) Type of data items SMALL, MEDIUM and LARGE,
(b) Reference variable names.
(c) Type of reference variables and their structure.
(d) Pointer type variables.
(e) Reference to fields TSHIRT and NEXTSIZE.
(f) Which are the static and dynamic variables?

14.14. Refer to Exercise 14.13. Explain the action of the following statements :

(a) **if** MEN = BOYS **then** MEN := **nil**
(b) **if** MEN↑ = BOYS↑ **then** MEN := **nil**
(c) **if** BOYS = **nil then** MEN ↑. TSHIRT := LARGE ;
(d) **if** MEN ↑. TSHIRT < > SMALL **then** MEN ↑. TSHIRT := MEDIUM ;

14.15. What is a circular linked list? Design a program to generate it.

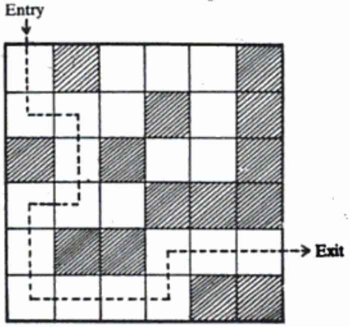
14.16. Draw the doubly linked circular list and prepare a Pascal program to create it.

14.17. Explain the differences and similarities between linear arrays and linked lists.

4.18. Develop a program (a) to concatenate two linked lists, (b) to divide a linked list in two linked lists.

14.19. Develop a program to find the path through a maze.

A maze is a rectangular array of cells, coloured white and black, as shown in the adjoining figure:



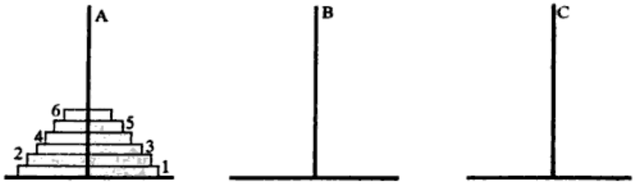
There are designated entry and exit cells. Movement from one white to an adjacent (horizontally or vertically) white cell is allowed. Generally, mazes have multiple paths from the entrance cell, with all but one of them terminating in a dead end. You are required to find one path that reaches the exit successfully. The reader should appreciate that the basic problem is to search for a solution among several alternatives.

14.20. Develop an algorithm and design a program, via stacks and pointers, to sort the following data.

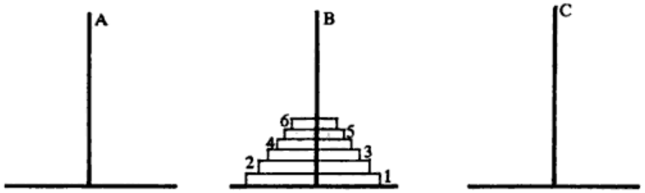
2, 6, 1, 8, 9, 2, 13, 4, 7, 8

in descending sequence by Quicksort procedure.

14.21. 'Towers of Hanoi Game' is played with three rods A, B, C and a certain number of disks of varying diameters having holes in the centre. The disks 1, 2, 3, 4, ... are put on a



rod in order of decreasing size,making a sort of tower. This is known as 'Tower of Hanoi', and, hence the name of the game. The purpose of the game is to transfer the tower on rod A to rod B, making use of rod C as an intermediary. Finally, the arrangement should appear as



shown above. The rules of the game are: (i) only one move is to be made to take a disk from the top of one tower and place it on another rod to form another tower, (ii) it is not allowed to put the disk on the ground or above another smaller disk.

Algorithm for this game may be developed either by iteration or recursive techniques. Design prcgrams for these algorithms using pointers and compare their efficiency. (There is a surmise that the game of Towers of Hanoi was first played by Pandits of Benaras).

Chapter 15

Structured Program Design Concepts

Computer program design refers to developing programs using a programming language and certain techniques which help to prepare programs in a consistent way that are easy to write and understand. Moreover, the program should be modifiable, maintainable with resonable efforts and upgradable as well. If the problem is small and simple, preparing the program is straightforward, however, for complex and difficult problems, programs are lengthy and involve lot of effort for their development.

It is always an involved but creative process to design good programs and software systems. Program development has become very expensive and attempt is always made to write programs which are general in design and portable. Testing, debugging and implementation of programs should involve minimal effort as far as possible. To achieve these objectives, several program design strategies have been suggested and used in practice. Commonly used methodologies have been modular design, structured programming, top-down approach, bottom-up technique, and so on. These techniques have been used to meet the above design goals of a software system and have better productivity of programmers. Such studies form part of the Software Engineering field in Computer Science. Here, we shall present a brief overview of the modular and structured design techniques for developing programs and urge the reader to supplement the presentation by making reference to a book on Software Engineering.

Program development strategies are not unique but only empirical approaches. You may come across some variations as well. However, the present introduction will help you do appreciate and use these techniques while developing large programs and software packages.

In the earlier chapters, we introduced concepts of algorithms, flowcharts and program development. Flowcharts constitute a pictorial language and are easy to understand and prepare. There has been another mode of representation/expression of algorithms. This is the pseudocode representation. Such codes have been popularly used to denote the control structures of structured programming, algorithms of problems, and so on. The pseudocode representation enables the user to express his ideas about program logic in a natural, English-like form. We shall discuss first pseudocode representation and then go over to the study of modular and structured programming.

15.1 Pseudocode

We have seen that the algorithm of a problem can be expressed either in

English or as a flowchart. The algorithm is translated into the programming language for solving the problem on the computers. There is another notation for expressing the algorithm, and this is by using English words/sentences and commonly used computer programming language words such as IF, THEN, ELSE, DO, WHILE, REPEAT, STOP, END, OTHERWISE, and so on. Such a notation is referred to as a Pseudocode. The pseudocode of a problem solution is somewhere in between English and the programming language. We illustrate this by an example.

When it is cold, wear woollen shirt, however, if it is not, wear cotton shirt.

This statement can be expressed as:

```
IF cold,
   wear woollen shirt
OTHERWISE
   use cotton shirt ;
```

The latter statement may be said to be in pseudocode.

As another example, consider a right quadrilateral having its adjacent sides as A and B. If A=B, then it is a square, if $A \neq B$, then it is a rectangle. We can express this in a pseudocode as

```
IF sides A and B are equal,
   THEN  the right qudrilateral is square
   OTHERWISE it is a rectangle ;
```

Alternately, the pseudcode may also be written as

```
IF side  A = side B
   THEN  square
   ELSE  rectangle ;
```

or a variation of this.

All symbols used in a pseudocode must be defined and properly documented so that the code can be understood by any other person.

We shall indicate the end of a statement or step in a pseudocode by a semicolon (;), though you may come across some different notation as well in literature.

The data assignment concepts and rules, as discussed in Section 2.2, are also applicable to variables used in pseudocodes, and we shall use them to prepare codes for various examples.

We can express the complete solution of a problem in pseudocode. As a simple illustration, we take the example of evaluating the area of a rectangle when its length and breadth are given. Its pseudocode may appear as :

```
READ  data for variables  Length, Breadth ;
COMPUTE  Length * Breadth ;
STORE  the product in variable Area ;
PRINT  the value of Area ;
```

or as

```
SET variables Length and Breadth to initial values ;
MULTIPLY Length and Breadth ;
STORE the product in variable Area ;
PRINT Area ;
```

or as

```
READ Length, Breadth ;
Area ← Length * Breadth ;
PRINT Area ;
```

and so on.

There are no hard and fast rules to write pseudocodes. It is a notation that allows us to express the logic of problem solution in a somewhat formalized way without being familiar with the syntax of a specific programming language. There are no formal syntactical rules to remember to prepare pseudocodes.

The pseudocode clearly describes the function to be performed, how to be performed, and can be directly translated into any of the computer programming languages. The advantages of preparing pseudocode are :

- a convenient way to code the problem solution
- translation into any programming language is straightforward
- documentation at every stage is automatic
- changes into the code can be made easily
- provides a detailed description of the complete source program
- thought processes can be better expressed
- better co-ordination between different programs as the code is in English-like language.

The pseudocode has also been termed as a Program Development Language (PDL) which is at a level higher than the programming language. However, it differs from computer programming in following respects :

(a) There are no formal rules to prepare pseudocodes.
(b) Operations can be specified at any level of simplicity or complexity. For example, statements of the type
  - Compare the velocity of two objects
  - Economic factor $= \sqrt{y} + a/b + (c(\tan\theta + \ln(x+y)))$

  can be freely used in pseudocodes.
(c) Pseudocode is much easy to understand while the source program (written in a computer programming language) may not be so.

Flowcharts are the pictorial representation of algorithms. We can prepare flowcharts for pseudocodes and vice versa. In our subsequent discussion, we shall illustrate the concepts of modular and structured programming using flowcharts and give pseudocodes as well for illustration.

15.2 Modular Design

Modular design refer to the division of the entire program into subtasks or modules. A module may be defined as a logically self-contained unit of a larger program. Thus, a complete program is a collection of modules which have been integrated suitably to achieve the desired objectives. Every module is expected to have the following characteristics :

- contain instructions and processing logic
- is distinct and logically separate
- performs a well-defined task in a program
- has one-entry and one-exit
- can be tested, debugged and compiled separately
- can use other modules
- is of general design and can be integrated with different kinds of appropriate programs/software systems.

There may be some violations but these provide the general guidelines to design modules. Examples of modules are procedures, functions, subroutines, Pascal blocks, a collection of instructions which perform a well-defined task, such as input/output of data, validation of data, and so on. A module should normally consist of 25-50 lines of program code. Smaller modules are, generally, a waste of time, while larger modules may be difficult to debug and integrate into a complete program. Attempt should be made to develop modules in a general way, so that they can be integrated with various programs and run on different machines.

An illustrative example of developing a software system using the modular approach is the design of a software package for implementing the various matrix operations or function integration routines. Recall, that the various operations which can be performed on matrices are :

- addition, subtraction, multiplication, division, transpose, inversion, finding the norm of a matrix, ...

A function/procedure subprogram can be developed for each operation. This will be a module. Similarly, code can be written for reading of data, checking of data and writing of data. Program codes for each operations may also be referred to as a module. They can be subprograms by themselves. The complete program for the implementation of all matrix operations can be modularized in this way. Example 10.3 illustrates the modular design strategy as given above.

At times, it may be difficult to decide as to how to divide the program into modules and how to put them together. There are no hard and fast rules for this and we learn much from experience. There are several advantages of writing programs in modular form :

- A single module is easier to write, debug and test than the entire program at a time.
- A module may be used in other programs as well, if it is prepared in a

general way and performs a common task. Thus, a library of standard and most commonly used modules can be prepared.

- Modifications can be incorporated in a single module with much less effort than in the entire program.
- Bugs can be easily located and isolated in a single module.
- A team of programmers can be employed to develop a complete system by modularization. Each programmer may be assigned the task of developing independent modules. The progress of a program, when written in modular form, can be assessed more easily.
- The interaction between parts of a program can be restricted to the interaction between the modules. This considerably simplifies the understanding of the program and its working.
- Maintenance of a program becomes easy when written in modules.

Modular design of software has advantages, no doubt, but there are some disadvantages as well. These are :

- Fitting the various modules into one program may be a difficult task, especially when different people are working on different modules.
- Debugging and testing the modules separately may not be easy, because other modules may produce the data used by the module being debugged. This necessitates the writing of "driver" programs which produce sample data and test the programs. These driver programs need extra programming effort.
- At times, it may be difficult to modularize programs in a reasonable way. If a program is modularized poorly, integration of modules will be a tough task.
- Generally, modular programs need extra time and memory, because the separate modules may repeat certain functions and involve some overheads as well.

Thus, the reader should appreciate that while modular design is certainly an improvement over conventional way of writing the entire program from the start, it does have some drawbacks as well. However, in spite of this, modular design has been playing an important role in the development of large software projects.

15.3 Structured Programming

In modular design, we develop the complete program in modules and then integrate them to form a complete unit. Next, the question arises as how to develop modules to keep them distinct, easily understandable, modifiable and prevent them from interacting with one another. Further, how to ensure the clear and concise sequence of operations in a module and isolate errors.

One possible answer to this has been proposed by Dijkstra. He advocated that the use of unconditional transfer statements (e.g. goto) in a program should be eliminated. He further reported—"the ease of reading and undrestanding program listings becomes inversely proportional to the number of unconditional transfers of control which they contained". This has been found to be quite plausible and due to this, modern practices of program development insist that the use of goto statements in a program must be kept to a minimum or avoided completely as far as possible. It has been proved that every program, however, complicated can be rewritten in an equivalent form using only three types of basic structures :

(1) sequential
(2) selection
(3) iteration

Each of these constructs has only one entry point and one exit point. We can design flowcharts, pseudocodes or program using these three constructs. They are called respectively structured flowcharts, structured pseudocodes and structured programs. We shall refer to structures (1), (2), (3) as the basic logic structures or constructs in our discussion. Design and use of these logic constructs is explained below.

15.4 Basic Structured Constructs

(i) *The Sequential Construct*

When statement(s)/modules are executed in sequence, they are said to form sequential construct. A flowchart for this is shown in Fig. 15.1 There is only one entry and one exit point.

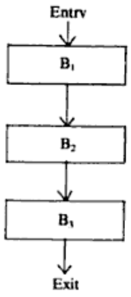


Fig. 15.1: Sequential Construct

Here B_1, B_2, B_3 denote block names. A block may consist of one or more statements. When a block name is enclosed in a rectangle, it would imply 'execute' or 'process' the block. Thus B_1 implies the execution of block B_1. The pseudocode for Fig. 15.1 may appear as

```
execute $B_1$ ;
execute $B_2$ ;
execute $B_3$ ;
```

statements while some condition is satisfied. Diagramatically, it can be shown as given in Fig. 15.3.

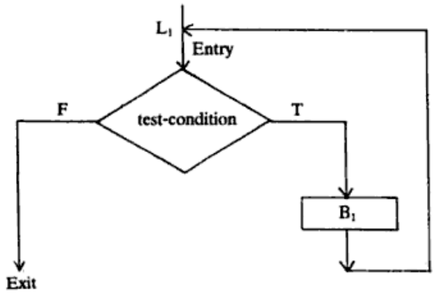


Fig. 15.3: Iterative Construct

Here, as long as the test-condition is true, block B_1 is executed, otherwise exit is made. The iteration construct also has single entry and single exit.
The pseudocode for iterative construct may be designed as :

```
WHILE test-condition TRUE DO
  execute $B_1$ ;
```

The use of word DO implies that the block following it is to be executed repeatedly.
An example of this construct from Pascal is :

```
while C > 0 do
  begin
    F := A+B ;
    D := A−B ;
    writeln (F, D)
  end ;
```

The repetitive construct has been implemented via repeat-until statements as well. It has also single-entry and single-exit and its flowchart is shown in Fig. 15.4.

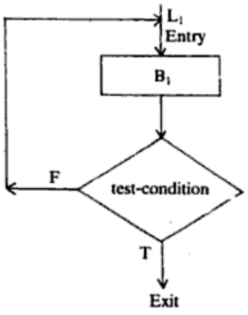


Fig. 15.4 : Repeat-Unitil-Construct

The pseudocode for Fig. 15.4 may be as :

```
L₁ : execute B₁ ;
   IF  test-condition  FALSE
      THEN  goto L₁
               ELSE  exit ;
```

Here L_1 has been used as a label.

However, a more appropriate code is

```
REPEAT
   execute B₁

      UNTIL  test-condition TRUE ;
```

Both the above iterative constructs have been used by programmers, though WHILE-DO was suggested originally.

The three basic constructs may be combined to generate bigger designs. An example is given in Fig. 15.5.

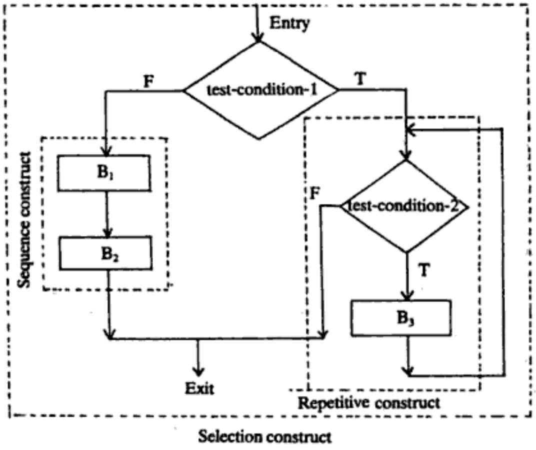


Fig. 15.5: Combination of logical constructs

The complete logic of this figure has a single-entry and single-exit. The pseudocode may be developed as

```
IF  test-condition-1  TRUE
   THEN (WHILE test-condition-2 TRUE DO execute B₃ ;)
      ELSE  execute B₁
               execute B₂ ;
```

15.5 Structured Programs

Programs designed using the three constructs—sequence, selection and iteration—are said to be structured programs and the approach is called Structured Programming. We shall illustrate and implement the program development process using structured programming constructs via flowcharts as they are more transparent, instructive and easy to understand. Actual writing of a program in a computer programming language, corresponding to the flowchart, is then a straightforward process.

We have seen that the valid combination of the logical constructs leads to structured programs. Any violation of this leads to unstructured programs. We examine both the unstructured and structured program designs and illustrate the conversion of unstructured programs to structured ones in the following.

Let there be five blocks of B_1, B_2, B_3, B_4, B_5 and we want to execute them in the sequence

$$B_1 \rightarrow B_4 \rightarrow B_3 \rightarrow B_2 \rightarrow B_5 \rightarrow B_3 \rightarrow B_4 \rightarrow B_5$$

One way of arranging their execution may be as shown in Fig. 15.6(a). (Numbers indicate the order of flow of execution).

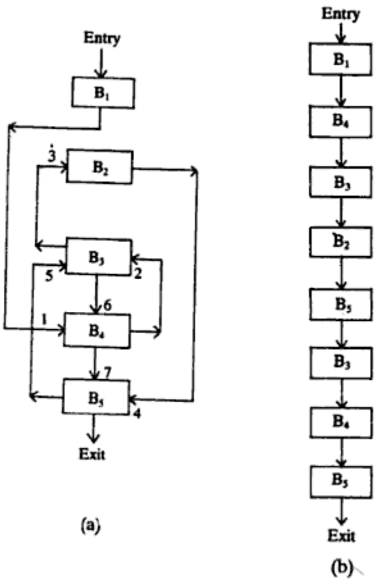


Fig. 15.6: (a) Unstructured and (b) structured design of sequence constructs

The arrangement of Fig. 15.6(a) is unstructured because there are double entries to blocks B_3, B_4 and B_5. Moreover, there is unrestricted unconditional transfer of execution control among the blocks. This makes it very difficult to keep track of program flow. Structured design demands that there should be single-entry and single-exit to the blocks and minimal use of unconditional transfers.

The structured version of Fig. 15.6(a) is shown in Fig. 15.6(b). Now there is single-entry single exit for every block, but execution of blocks B_3, B_4, B_5 has been repeated. This duplication of codes becomes essential many times while writing structured programs and is a drawback of this methodology. But the advantage of structured design is the ease of understanding the program.

Another example of unstructured diagram is given in Fig. 15.7.

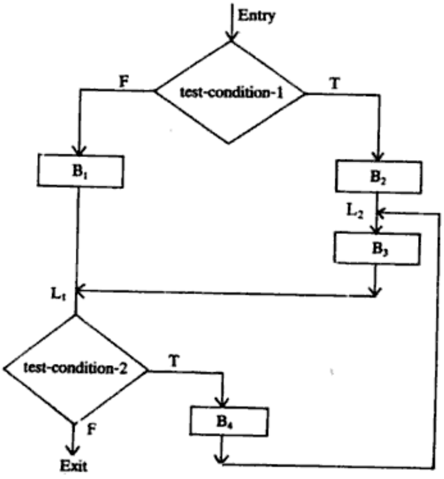


Fig. 15.7: Unstructured diagram

Here, you may notice that block B_3 has double entry and there are two entries at L_1 as well.

The pseudocode for Fig. 15.7 may appear as :

```
.............
    IF test-condition-1 TRUE
      THEN execute B2
        L2 : execute B3
          GOTO L1
      ELSE execute B1 ;
        L1 : IF test-condition-2 TRUE
          THEN execute B4
            GOTO L2
          ELSE next-step ;
```

The student should appreciate the presence of forward and backward jumps in this code.
The structured version of Fig. 15.7 may appear as :

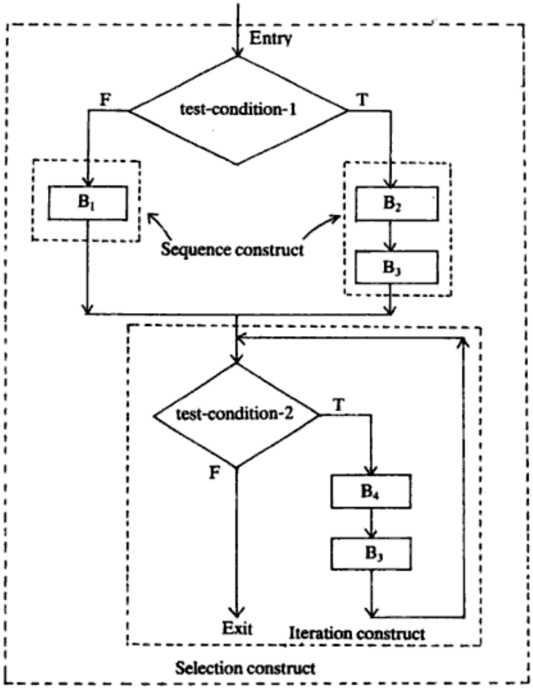


Fig. 15.8: Structured version of Fig. 15.7

Now block B_3 has been duplicated to avoid unconditional back transfer from block B_4 to B_3. This keeps the program execution flow in the forward direction.
The pseudocode for Fig. 15.8 is :

```
IF test-condition-1 TRUE
   THEN execute B2
           execute B3
   ELSE execute B1 ;
WHILE test-condition-2 TRUE DO
   execute B4
   execute B3 ;
..............
..............
```

A further example of unstructured program may be as given in Fig. 15.9.

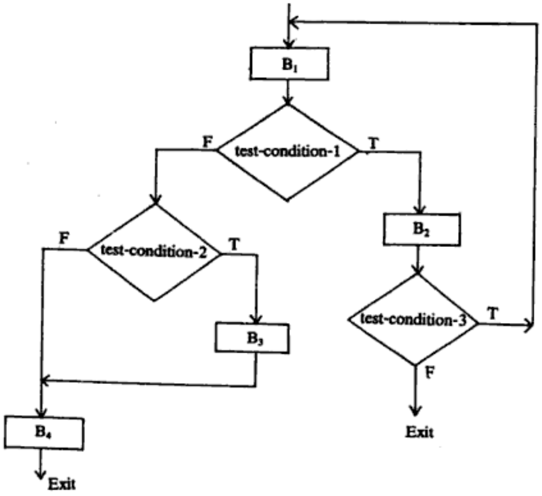


Fig. 15.9: Unstructured diagram with multiple exits

Do you know why it is unstructured?

15.6 Extension of structured constructs and use of goto statement

As mentioned before, any program can be developed using the three basic constructs—sequence, decision and repetition. However, the case construct is often included as the fourth permissible logic structure for preparing programs. This construct allows multibranch in a program. Its representation is given in Fig. 15.10.

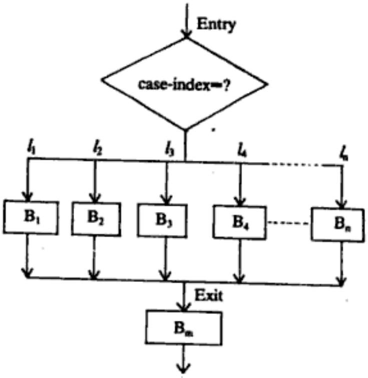


Fig. 15.10: Case Construct

If case-index equals

l_1 block B_1 is executed
l_2 block B_2 is executed
.
.
.
l_n block B_n is executed

After the execution of any of the blocks $B_1, B_2, \ldots. B_n$, exit is made and control goes to the following block B_m.

We have already studied the format and use of **case** construct in Pascal. While developing structured programs in Pascal, you may use the following constructs : if-then, if-then-else, case, for-do, while-do and repeat-until. The structures if-then and for-do are special cases of selection and repetitive constructs. This flexibility offers more convenience to the programmer to write purely structured programs.

Structured programming has also been termed, sometimes, as 'goto-less' programming, but now this concept has changed. Appropriate use (though limited) of goto statement may be made along with the other structured programming constructs to design programs which are well designed and easy to understand. While using goto statement, control should generally be passed on in the forward direction as far as possible. Most of the modern programming languages, such as Pascal, Modula-2, Ada, Fortran 77, Basic, Cobol, C, provide enough language constructs to prepare structured programs. Attempt should always be made to develop programs following structured programming methodology.

Example 15.1

We develop two programs for solving simultaneous equations by Gauss-Jordan† method. Program 15.1(a) has been designed in a conventional way without any consideration for use of structured constructs or goto statements. This is an unstructured programs. Program 15.1(a) uses the constructs of structured program design and try to keep the use of goto statements to a minimal.

Program 15.1 (a)

```
program GAUSSJORDAN (input, output) ;
{ Unstructured version of the program } ;
label 100, 200, 300, 400, 500, 600 ;
var
  I, J, K, L, M, N, NPLUS1, NMINUS1 : integer ;
  A : array [ 1 .. 10, 1 .. 10] of real ;
  X : array [ 1 .. 10] of real ;
  TEMP : real ;
```

†Algorithm for Gauss-Jordan method is standard and may be found in a book on Numerical Analysis.

```
    { Read in the system of equations }
    { Right hand side of the equations is the (n + 1)th
         column of the matrix }
    for I := 1 to N do
      begin
        for J := 1 to NPLUS1 do
          read (A [I,J] ) ;
          readln
      end ;
  { Print the system of equations }
    for I := 1 to N do
      begin
        for J := 1 to NPLUS1 do
          write (A [I, J] ) ;
          writeln
      end ;
  { Main program segment }
  K : = 1 ;
  { Check if the diagonal element of the matrix is zero }
  while (K <= N) do
  begin
    if (A [ K, K ] = 0.0) then
    begin
      FLAG := true ;
      L := K + 1 ;
      for I := 1 to N do
        begin
          if ((A [I, K] < > 0.0) and FLAG) then
          begin
            for J := 1 to NPLUS1 do
            { Swapping of rows is done here }
              begin
                TEMP := A [I, J] ;
                A [I,J] := A [K, J] ;
                A [K, J] := TEMP
              end ;
            FLAG := false
          end
        end
  end ;

if (FLAG) then
  begin
    writeln ('The method fails for this set of equations') ;
    goto 600  { This statement has been used to exit from the program }
  end
  else { If diagonal element is not zero, proceed }
```

```
         begin
           TEMP := A [K, K] ;
           for J := 1 to NPLUS1 do
             A [K, J] := A [K, J]/TEMP ;
           for I := 1 to N do
             begin
               if (I < > K) then
                 begin
                   TEMP := A [I, K] ;
                   for J := K to NPLUS1 do
                     A [I, J] := A [I, J]—A [K, J] * TEMP
                 end
               end
             end ;
           K := K+1
         end ;
       { Print the solutions }
     writeln ('The solution is' } ;
     for I := 1 to N do
       writeln ('X [', I, '] = ', A [I, NPLUS1] ) ;
   600 :
   end.
```

Let us compare the two versions of the GAUSSJORDAN program as given in 15.1(a) and 15.1(b). Flow of program execution in the Main program segment is indicated by arrows. The flow moves forward and backward haphazardly in version 15.1(a). This makes program debugging very difficult. It would have been far better if the program flow was in the forward direction and there is no 'criss-crossing'. Program 15.1(b) avoids this. Use of **goto** statements has been eliminated, except one, by the proper use of structured constructs while-do, for-do, if-then and if-then-else. The **goto** statement has been used only once to exit from the program. Such use of **goto** statements has been recommended as it helps to make the program more compact and clear. We could have avoided this use of **goto** statement as well by using appropriate programming techniques, but that would make the program unnecessarily long and complex. Judicious use of **goto** statement will always help to write more efficient and transparent programs.

15.7 Structured Modular Programming

We know that the complete computer program of a software project may consist of many modules. The whole program may be structured with respect to its modules as well. We illustrate the conversion of an unstructured modular program to a structured modular program in the following. It is assumed that each program module has been written using the constructs of structured programming.

Suppose that a program has been split into modules and these modules have been organized as shown in Fig.15.11. Numbers 1, 2, . . . , 9 indicate the modules (We had used the notation $B_1, B_2, B_3, \ldots$ to denote blocks).

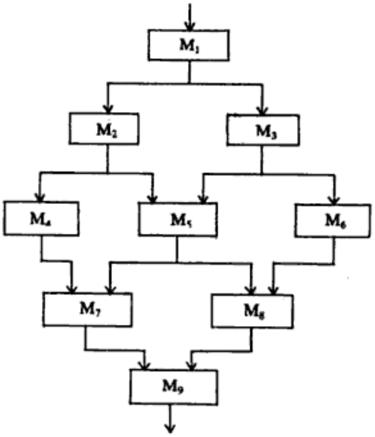


Fig. 15.11: Unstructured program

This program is unstructured because each box does not correspond to the one-entry one exit rule. There is more than one 'branch' (each arrow indicates branching) to the boxes. Let us see what type of difficulties may be encountered in such a program.

Consider module 5. We can enter this module from module 2 and module 3. Its correct execution depends on the various things such as constants, variables, statements and some other conditions, depending on its design. Values of variables needed in module 5 may have been defined in modules 2 and 3. we cannot determine the execution of module 5 without knowing what has happened in module 2 and 3. The situation is much more complicated in succeeding modules. For instance, consider module 9. Here there are many possible paths of entry. Execution of module depends as to what happens in the previous modules, i.e. in modules 7 and 8 which further depends on what happens in modules 1, 2, 3, 4, 5 and 6. Now, when there is some problem in module 9, we cannot locate the bug easily, as the bug may have been in one of the earlier modules from where execution has reached module 9.

In order to convert the unstructured program of Fig. 15.11, we duplicate those modules that may be entered from more than one place. Fig. 15.11 may be restructured as shown in Fig. 15.12. Now module 5 has been duplicated while modules 7 and 8 have been triplicated to ensure the condition of single-entry single-exit for the modules as well.

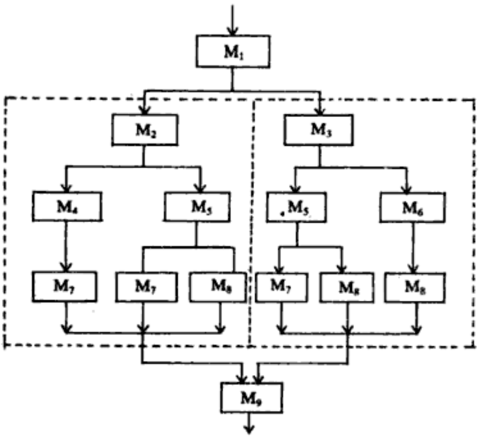


Fig. 15.12: Structured modular program

Now each module has one-entry one-exit, except the last module 9, which could have been simply the end of the program or it may also be duplicated.

There is clearly a disadvantage of duplicating the modules/codes. It requires more memory than the original unstructured design. If the modules/codes are small, it is well worth the cost of duplicating the code to generate a structure that can be broken as shown in Fig. 15.12. If the modules involve a substantial amount of coding, e.g. 50 or more statements, then the problem may be solved by making them callable procedures. but it is important that they be developed as procedures with formal arguments, so that their correctness can be determined without regard to the context in which they are executed. If this approach is taken, we will have multiple calls to a single copy of a procedure. This procedure also involves certain amount of overheads and hence inefficiency. Conversion of unstructured diagrams of Fig. 15.11 to structured diagram of Fig. 15.12 has followed one particular way. You may think your own procedure for conversion and implementation of structured methodology. Recall that the structured design approach is not unique and there can be enough variations, though we must try to adhere to the use of structured constructs and the single-entry single-exit principles as far as possible.

15.8 Advantages and Disadvantages of Structured Programming

We have seen that structured programs are written using certain constructs which are fairly commonly accepted. However, structured programs have their own advantages and disadvantages. These are summarized below.

(g) The single-entry single-exit rule is essential for structured programming. (T/F)
(h) A structured program should never contain a goto statement. (T/F)
(i) Structured programming can implement multibranch situations in a program. (T/F)
(j) The logic of all programs cannot be expressed as a combination of three basic logic constructs. (T/F).
(k) A structured program is always free of logic errors. (T/F).
(l) The case structure helps to implement multibranch situations in a program. (T/F)
(m) An unstructured program cannot be transformed into a structured form. (T/F)
(n) Structured or goto-less programming are same. (T/F)
(o) Structured programs may involve repetition of statements. (T/F)
(p) Modular and structured programming techniques are exact. (T/F)

15.2. Complete the following sentences :

(a) Pseudocode is at a level .... than the common programming languages.
(b) Pseudocode representation has .... commonly accepted .... rules.
(c) A single module is easier to .... and .... than the entire program at a time.
(d) It is always preferable to have the module length as of .... lines.
(e) Functions/procedures are examples of ....
(f) Every module of a software system must be designed using the logic constructs of .... programming.
(g) The basic logic constructs for preparing structured programs are ...., ... and ....
(h) A module must be .... and .... separate.
(i) Modular and structured programs generally need .... memory space.
(j) The productivity of a programmer is improved by using .... and .... programming techniques.
(k) Modification of .... programs is very time consuming.
(l) Maintenance of programs becomes easy when written in .... and .... form.
(m) Pascal is among the most suitable languages for designing .... programs.
(n) All languages may not have .... constructs normally used to design .... programs.

15.3 What do you understand by pseudocode? Illustrate by an example. Give advantages of preparing pseudocodes for problem solving.

15.4. Develop a pseudocode for deciding whether the given calendar year is a leap year or not.

15.5. Design a flowchart and a pseudocode to compute the roorts of a quadratic equation, taking due care for complex roots and trivial solution.

15.6. Explain the concept of a module. What should be its desirable features?

15.7. What do you understand by structured programming? Explain the meaning and significance of various structures used to design structured programs.

15.8. Prepare a summary of formats of basic logic structured constructs as available in the following languages : Ada, Modula-2, Fortran 77, Basic, C, Cobol, Algol, APL

15.9. Bring out the differences between modular and structured programming. Describe their advantages and disadvantages.

15.10. Study the programs of the worked examples in the book. List the structured and unstructured programs. Transform the unstructured programs to structured design.

15.11. Diagram of Fig. 15.9 is unstructured. Why? Prepare its structured format and write pseudocodes for both versions.

15.14. Look at the flowchart of Fig. 15.14 carefully:

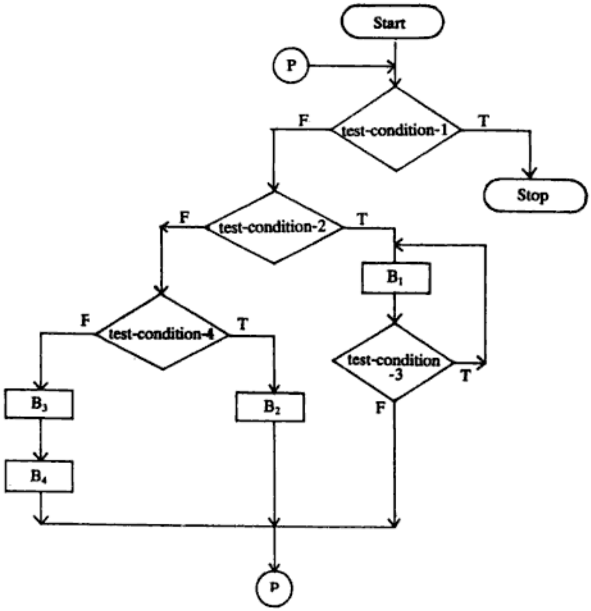


Fig. 15.14

Does it represent a structured or unstructured program? Prepare its pseudocode.

15.15. Refer to Exercise 15.14. Choose your own realistic sample for the blocks B_1, B_2, B_3, B_4. Code the program in Basic, Fortran 77 and Pascal. Run the programs on your system. Study the compactness and efficiency of the three implementations.

15.16. Using the principles of modular and structured programming, develop a software package, in Pascal, to evaluate the following integral :

$$\int_a^b f(x)\, d\, x$$

by (i) Rectangle rule, (ii) Trapezoidal formula (iii) Simpson's rule (iv) Newton's three-eigths rule (iv) Romberg's procedure.
(N.B. Algorithms for these methods may be found in books on Numerical Analysis). Your program should also estimate errors present in each method.
Use this package to compute the integral

$$\int_0^1 \frac{\sin\ x}{x^2+1}\, dx$$

accurate to 10^{-4} by various methods.

15.18. Develop a software package, in Pascal, using the modular and structured programming concepts to perform the following string operations :

- Reads a string of text.
- Computes the length of the complete string.
- Searches and replaces (if needed) a given substring within the string.
- Concatenates two strings.
- Finds the location of a substring in the string.
- Computes the frequency of occurrence of a given word in the string.
- Finds the number of vowels which have been used independently in the sentence, such as A and I.
- Calculates the length of each word in the sentence and tells the number of words with length equal to 1, 2, 3, . . . , 25.
- Matches two strings.
- Arranges the words of the string in alphabetical order.
- Locates the presence of punctuation symbols and finds their totals.
- Replaces the given word/character by the desired word/chracter.
- Doubles/halves the spacing between consecutive words.
-

(N.B. A word processing package or word processor is designed on the same pattern, though it may offer many more capabilities, such as screen formatting, margin adjustments, and so on.)

15.19. Which structure is used for multibranch situation in a program? Explain its function. Study the format and implementation of this construct in Pascal, Cobol, Fortran 77, Basic and C languages.

15.20. Sometimes, it is said that structured programming is "goto-less" programming. Would you agree with this statement? If not, illustrate your answer by an example where use of goto statement may be essential to design an efficient and compact program.

Appendix I

Pascal Operators and their Precedence

- Arithmetic

  + — * / div mod

- Boolean

  and or not

- Relational

  = <> < <= > >= in

- Operator Procedence (highest to lowest)

  not (highest)

  *** / div mod and**
  + — or
  = <> < <= > >= in (lowest)

There may appear two or more than two operators, at the same precedence level, in ah expression. In that case, successive operations are carried out from left to right.

Appendix II

Standard Identifiers and Pascal Reserved Words

(a) *Standard Identifiers*

abs	**arctan**				
boolean					
char	**chr**	**cos**			
dispose					
eof	**eoln**	**exp**			
false					
get					
input					
ln					
maxint	**minint**				
new					
odd	**ord**	**output**			
pack	**page**	**pred**	**put**		
read	**readln**	**real**	**reset**	**rewrite**	**round**
sin	**sqr**	**sqrt**	**succ**		
text	**true**	**trunc**			
unpack					
write	**writeln**				

(b) *Reserved Words*

and	**array**		**set**		
begin			**then**	**to**	**type**
case	**const**		**until**		
div	**do**	**down to**	**var**		
else	**end**		**while**	**with**	
file	**for**	**function**			
goto					
if	**in**				
label					
mod					
nil	**not**				
of	**or**				
packed	**procedure**	**program**			
record	**repeat**				

Appendix III

Pascal supplied Functions and Procedures

(a) *Functions*

In the following, we shall denote the argument of a function by **x**. (Function names have been organised alphabetically).

Function	*Argument type*	*Result type*	*Purpose*
abs (x)	integer/real	same as x	Finds the absolute value of x.
arctan (x)	integer/real	real	Computes arctangent of x.
chr (x)	integer	character	Determines the character represented by x.
cos (x)	integer/real	real	Evaluates the cosine of x; the argument must be in radians.
eof (x)	file	boolean	Determines whether an end-of-file mark is there.
eoln (x)	file	boolean	Determines whether an end-of-line mark is there.
exp (x)	integer/real	real	Computes e^x.
ln (x)	integer/real	real	Evaluates natural log of x ($x > 0$).
odd (x)	integer	boolean	Finds whether x is odd or even; if x is odd, value returned is true, otherwise false.
ord (x)	character	integer	Finds the integer number corresponding to argument x in ASCII representation.
pred (x)	integer/ character or boolean	same as x	Determines the precedessor of x.
round (x)	real	integer	Rounds the value of x to the nearest integer.
sin (x)	integer/real	real	Calculates the sine of x (x in radians).
sqr (x)	integer/real	same as x	Computes the square of x.
sqrt (x)	integer/real	real	Evaluates the square root of x ($x \geqslant 0$).

succ (x)	integer/ character/ boolean or enumerated	same as x	Determines the successor to x.
trunc (x)	real	integer	Truncates x to return integer value.

(b) *Procedures*

Name	*Argument type*	*Purpose*
dispose (x)	Pointer	Deletes the dynamic variables referenced by the pointer x.
get (x)	File	Advances file buffer to the next component and places the value of the component in the buffer.
new (x)	Pointer	Creates a dynamic variable that is accessed through pointer x.
pack (a, i, b)	a, b: arrays i: integer	Takes the elements starting at subscript position i of array a and copies then into array b, starting at the first subscript position, in the packed mode.
read (. . .)		Read data items from an input file, but do not skip to the next line.
readln (. . .)		Read data items from an input file, then skip to the next line.
reset (x)	File	Sets file x at the start for reading.
rewrite (x)	File	Prepares a file for writing.
unpack (b, a, i)	a, b: arrays i: integer	Takes the elements starting at the first subscript. position of packed array b and copies them into array a starting from position i.
write (. . .)		Write data items to an output file without skipping to the next line.
writeln (. . .)		Write data items to an output file, then skips to the next line.

statements, etc. in BNF are given below. They represent the most general definition and should be interpreted as has been explained above.

⟨letter⟩ : : = a|b| . . .|x|y|z|A|B|. . .|X|Y|. . .|Z
  ⟨letter or digit⟩ : : = ⟨letter⟩|⟨digit⟩
⟨identifier⟩ : : = ⟨letter⟩ { ⟨letter or digit⟩ }
  ⟨constant identifier⟩ : : = ⟨identifier⟩
  ⟨string⟩ : : = ⟨character⟩
⟨constant⟩ ::= ⟨constant identifer⟩|⟨unsigned number⟩| ⟨sign⟩ ⟨constant identifier⟩ | ⟨sign⟩ ⟨unsigned number⟩ | ⟨string⟩ | **nil**
⟨variable⟩ : : = ⟨variable⟩ | ⟨component variable⟩ | ⟨referenced variable⟩
  ⟨variable⟩ : : = ⟨variable identifier⟩
  ⟨variable identifier⟩ : : = ⟨identifier⟩
⟨function desginator⟩ ::= ⟨function identifier⟩ | ⟨function identifier⟩ (⟨actual parameter⟩ {, ⟨actual parameter⟩ })
⟨function identifer⟩ :: = ⟨identifer⟩
⟨actual parameter⟩ : : = ⟨expression⟩ | ⟨variable⟩| ⟨procedure identifer⟩ | ⟨function identifier⟩
  ⟨procedure identifer⟩ : : = ⟨identifier⟩
⟨factor⟩ : : = ⟨variable⟩ ⟨unsigned constant⟩ | (⟨expression⟩) ⟨function designator⟩ | *not* ⟨factor⟩|⟨set⟩
    ⟨set⟩ : : = [⟨element list⟩]
  ⟨element list⟩ : : = ⟨element⟩ { ⟨element⟩} | ⟨empty⟩
⟨element⟩ :: = ⟨expression⟩|⟨expression⟩ . . ⟨expression⟩
⟨empty⟩ : : =
  ⟨multiplying operator⟩ : : = *|/|**div**|**mod**|**and**
⟨term⟩ : : = ⟨factor⟩|⟨term⟩⟨multiplying operator⟩ ⟨factor⟩
  ⟨adding operator⟩ : : = + | − |**or**
⟨simple expression⟩ : : = ⟨term⟩ | ⟨sign⟩ | ⟨term⟩ |
  ⟨simple expression⟩ ⟨adding operator⟩ ⟨term⟩ | ⟨simple expression⟩ ⟨adding operator⟩ ⟨term⟩
  ⟨relation operator⟩ : : = < | ≤ | = | > | ≥ | > |**in**
⟨expression⟩ : : = ⟨simple expression⟩ | ⟨simple expression⟩ ⟨relational operator⟩ ⟨simple expression⟩
  ⟨statement⟩ : : = ⟨unlabelled statement⟩ | ⟨label⟩ : ⟨unlabelled statement⟩
  ⟨unlabelled statement⟩ : : = ⟨simple statement⟩ | ⟨structured statement⟩
⟨simple statement⟩ : : = ⟨assignment statement⟩ | ⟨procedure statement⟩ | ⟨go to statement⟩ | ⟨empty statement⟩
  ⟨empty statement⟩ : : = ⟨empty⟩
⟨assignment statement⟩ : : = ⟨variable⟩ : = ⟨expression⟩| ⟨function identifier⟩ : = ⟨expression⟩
⟨structured statement⟩ : : = ⟨compound statement⟩ | ⟨conditional statement⟩ | ⟨repetitive statement⟩ ⟨**with** statement⟩
⟨compound statement⟩ : : = **begin** ⟨statement⟩ { ; ⟨statement⟩ } **end**
⟨conditional statement⟩ : : = ⟨**if** statement⟩ | ⟨**case** statement⟩
⟨**if** statement⟩ :: = **if** ⟨expression⟩ **then** ⟨statement⟩ **if** ⟨expression⟩ **then** ⟨statement⟩ **else** ⟨statement⟩

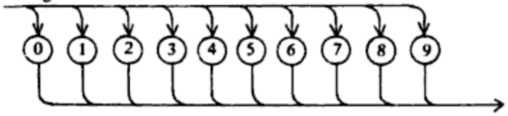
⟨**case** statement⟩ : : = **case** ⟨expression⟩ **of** ⟨case list element⟩ { : ⟨case list element⟩ } **end**
⟨case list element⟩ : : = ⟨case label list⟩ : ⟨statement⟩|⟨empty⟩
  ⟨case label list⟩ : : = ⟨case label⟩ {, ⟨case label⟩ }
  ⟨case label⟩ : : = constant
⟨repetitive statement⟩ : : = ⟨**while** statement⟩|⟨**repeat** statement⟩| ⟨**for** statement⟩
    ⟨while statement⟩ : : = **while** ⟨expression⟩ **do** ⟨statement⟩
    ⟨repeat statement⟩ : : = **repeat** ⟨statement⟩ { : ⟨statement⟩ } **until** ⟨expression⟩
    ⟨type definition⟩ : : = ⟨identifier⟩ = ⟨type⟩
⟨simpletype⟩ : : = ⟨scalar type⟩|⟨subrange type⟩ | ⟨type identifier⟩
  ⟨type identifier⟩ : : = ⟨identifier⟩
  ⟨scalar type⟩ : : = (⟨identifier⟩ {, ⟨identifier⟩ })
  ⟨subrange type⟩ : : = ⟨constant⟩ . . ⟨constant⟩
⟨constant definition part⟩ :: = ⟨empty⟩ | **const** ⟨constant definition⟩ { : ⟨constant definition⟩ }
  ⟨constant definition⟩ : : = ⟨identifier⟩ = ⟨constant⟩
⟨variable declaration part⟩ : : = ⟨empty⟩ | **var** ⟨variable declaration⟩ {, ⟨variable declaration⟩ }
⟨variable declaration⟩ : : = ⟨identifier⟩ {, ⟨identifier⟩ } : ⟨type⟩
  ⟨structured type⟩ : : = ⟨unpacked structured type⟩ | **packed** ⟨unpacked structured type⟩
⟨unpacked structured type⟩ : : = ⟨array type⟩ | ⟨record type⟩ | ⟨set type⟩ | ⟨file type⟩
  ⟨array type⟩ : : = **array** [⟨index type⟩ { ⟨index type⟩ }] **of** ⟨component type⟩
  ⟨index type⟩ : : = ⟨simple type⟩
  ⟨component type⟩ : : = ⟨type⟩
⟨component variable⟩ : : = ⟨indexed variable⟩ | ⟨field designator⟩ | ⟨file buffer⟩
⟨indexed variable⟩ : : = ⟨array variable⟩ | ⟨expression⟩ {, ⟨expression⟩ }].
  ⟨array variable⟩ : : = ⟨variable⟩
⟨for statment⟩ : : = **for** ⟨control variable⟩ : = ⟨for-list⟩ **do** ⟨statement⟩
    ⟨for-list⟩ : : = ⟨initial value⟩ **do** ⟨final value⟩
    ⟨initial value⟩ **downto** ⟨final value⟩
    ⟨control varaible⟩ : : = ⟨identifier⟩
    ⟨initial value⟩ : : = ⟨expression⟩
    ⟨final value⟩ : : = ⟨expression⟩
    ⟨record type⟩ : : = **record** ⟨field list⟩ **end**
⟨field list⟩ : : = ⟨fixed part⟩ | ⟨fixed part⟩ : ⟨variant part⟩ | variant part⟩

⟨fixed part⟩ : : = ⟨record section⟩ {, ⟨field identifier⟩ } : ⟨type⟩ | ⟨empty⟩
⟨variant part⟩ : : = **case** ⟨tag field⟩ ⟨type identifer⟩ **of** ⟨variant⟩ { ; ⟨variant⟩ }
  ⟨tag field⟩ : : = ⟨field identifier⟩ : | ⟨empty⟩
  ⟨variant⟩ : : = ⟨case label list⟩ : ⟨(field list⟩) | ⟨empty⟩
  ⟨field designator⟩ : : = ⟨record variable⟩ : ⟨field identifer⟩
    ⟨record variable⟩ : : = ⟨variable⟩
    ⟨field identifer⟩ : : = ⟨identifier⟩
  ⟨with statement⟩ : : = **with** ⟨record variable list⟩ **do** ⟨statement⟩

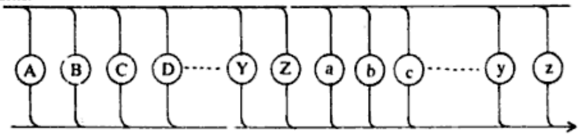
Appendix V

Syntax Diagrams

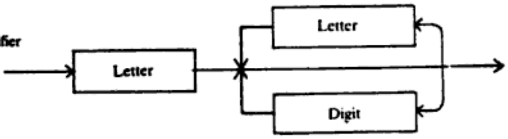
Digit



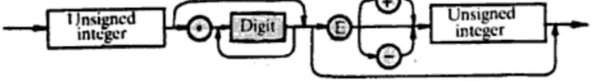
Letter

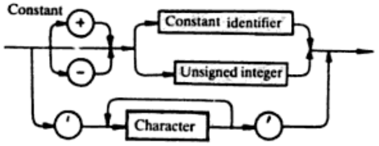


Identifier

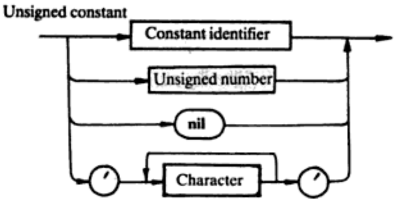


Unsigned number

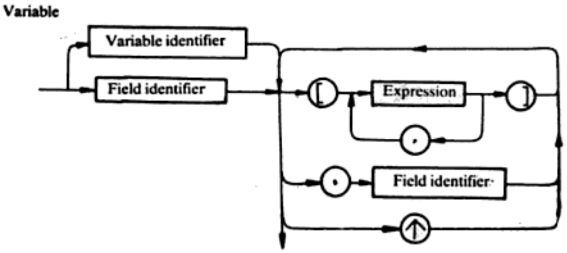




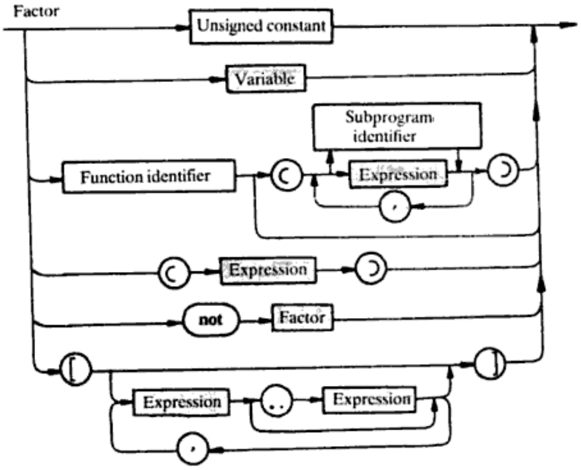
Constant
+
-
Constant identifier
Unsigned integer
'
Character
'

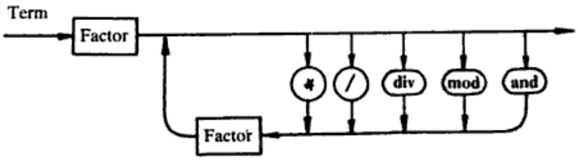


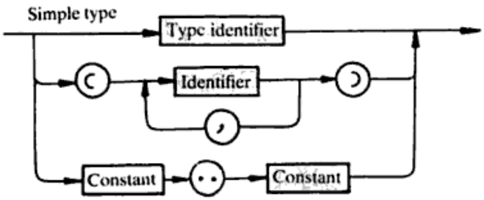
Unsigned constant
Constant identifier
Unsigned number
nil
'
Character
'



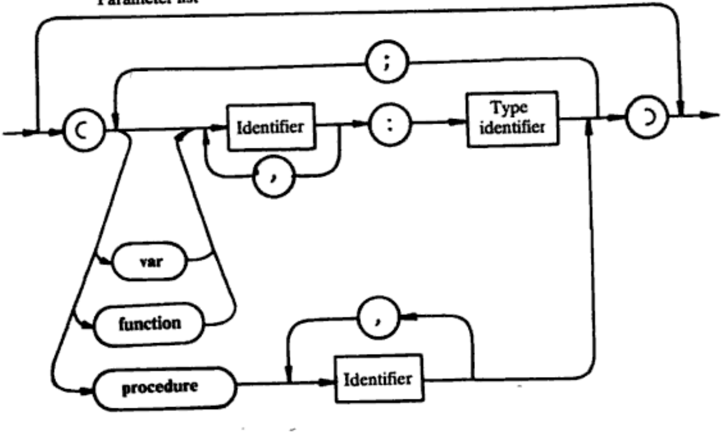
Variable
Variable identifier
Field identifier
[
Expression
]
,
.
Field identifier
↑


Factor
Unsigned constant
Variable
Subprogram identifier
Function identifier
(
Expression
,
)
(
Expression
)
not
Factor
[
Expression
..
Expression
,
]

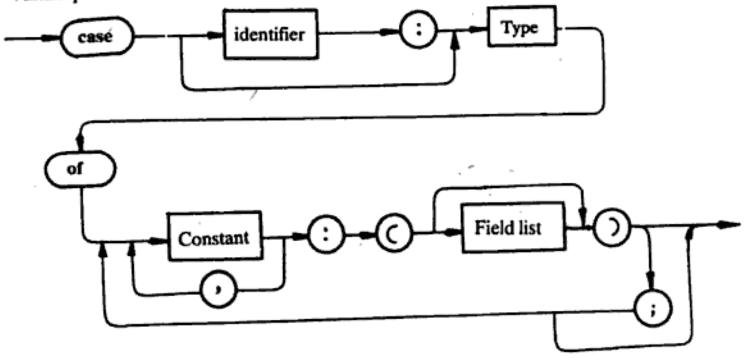

Term
Factor
*
/
div
mod
and
Factor


Simple type
Type identifier
(
Identifier
,
)
Constant
..
Constant

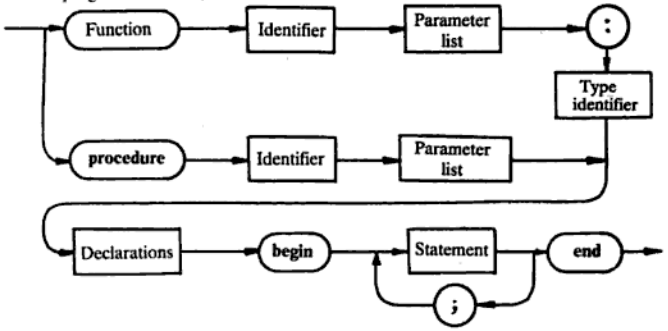
Parameter list



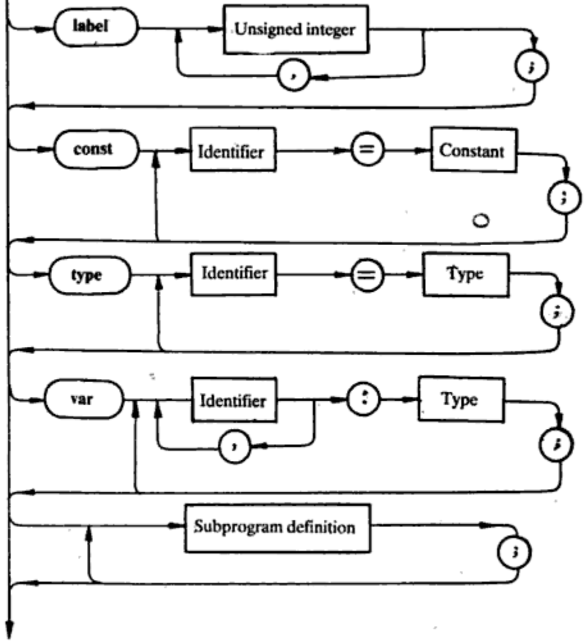
Variant part



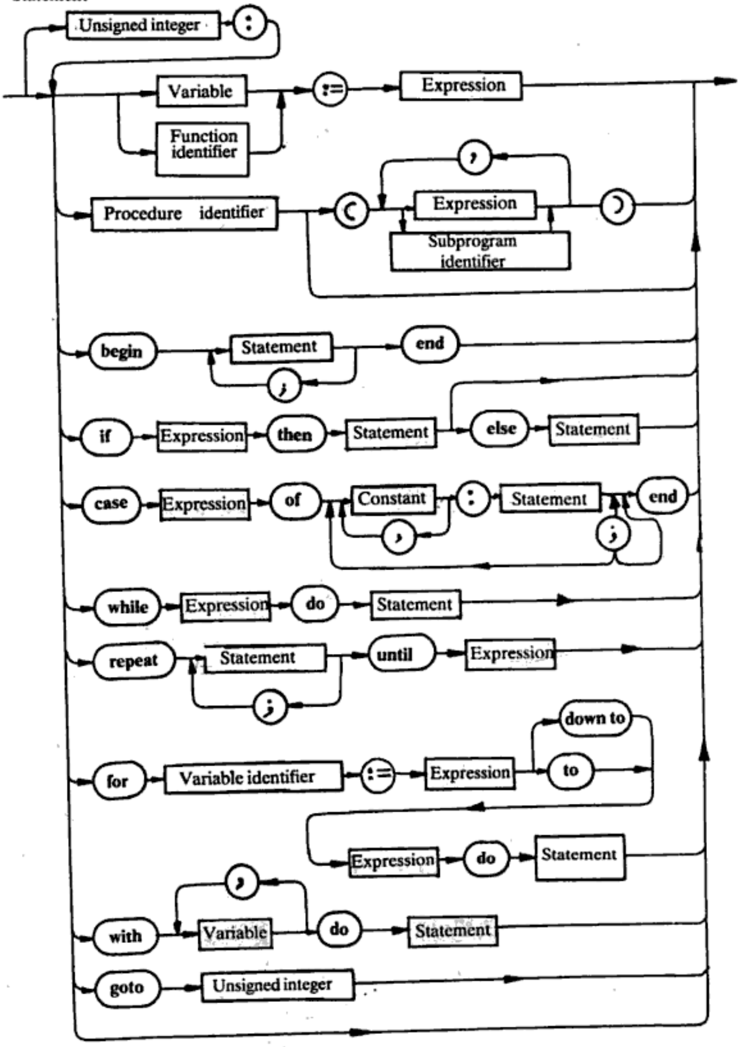
Subprogram definition



Declarations



Statement


Unsigned integer
:
Variable
Function identifier
:=
Expression
Procedure identifier
(
Expression
,
Subprogram identifier
)
begin
Statement
;
end
if
Expression
then
Statement
else
Statement
case
Expression
of
Constant
,
:
Statement
;
end
while
Expression
do
Statement
repeat
Statement
;
until
Expression
for
Variable identifier
:=
Expression
down to
to
Expression
do
Statement
with
Variable
,
do
Statement
goto
Unsigned integer

Appendix VI

ASCII and EBCDIC Character Sets

The word ASCII stands for Amerkan Standards Code for Information Interchange whereas EBCDIC denotes Extended Binary Code Decimal Interchange Code. These codes are commonly used to represent characters in computers. ASCII is used virtually in all microcomputers and several of others, while EBCDIC is popular in IBM computers.

Table A shows commonly used characters of ASCII character set. There is a numerical value associated with each character. It can be obtained from the sum of its row number and the column number. For instance, the ASCII value of A is 64 (row) + 1 (column) = 65; value for T is 80 (row) + 4 (column) and so on. The collating sequence of the characters is determined by the corresponding value associated with that character. Thus, in ascending sequence, characters appear as ƀ (32), ! (33), "(34), # (35), $ (36) . . . ., P (80), Q (81), ... , a (97), b (98), c (99), ... , z (122), ...

Table A : ASCII character set

0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16																
32	ƀ	!	”	#	$	%	&	’	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	↑	—
96		a	b	c	d	e	f	g	h`	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Table B shows a part of the EBCDIC character set. Now the lower case letters precede the upper case letters which in turn precede the numeric digits.

Table B : EBCDIC character set

0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16																
32																
48																
64	ƀ										¢	.	<	(	+	/
80	&										!	$	*	)	;	¬
96	–	/										,	%	–	>	?
112											:	#	@	'	=	"
128		a	b	c	d	e	f	g	h	i						
144		j	k	l	m	n	o	p	q	r						
160			s	t	u	v	w	x	y	z						
176																
192	{	A	B	C	D	E	F	G	H	I						
208	}	J	K	L	M	N	O	P	Q	R						
224	\		S	T	U	V	W	X	Y	Z						
240	0	1	2	3	4	5	6	7	8	9						

Bibliography

1. Wirth, N and Jensen, K.: Pascal-User Manual and Report (Springer-Verlag, 1975).
2. Wirth, N.: Systematic Programming (Prentice-Hall, 1973).
3. Dormey, R: How to Solve it on the Computer (Prentice-Hall, 1976).
4. Schneider, G.M. and Bruell, S.: Advanced Programming and Problem Solving with Pascal (Wiley, 1981).
5. Kernighan, B.W. and Plauger, P.J.: (a) The Elements of Programming Style (McGraw-Hill, 1974).
   (b) Software tools in Pascal (Addison-Wesley, 1981).
6. Singh Bhagat and Naps T.L.: Introduction to Data Structures (Tata McGraw-Hill, 1986).
7. Zeigler, C.A.: Programming System Methodologies (Prentice-Hall, 1983).
8. Kruse R.L.: Data Structures and Program Design (Prentice-Hall, 1987).
9. Dijkstra, E.W: A Discipline of Programming (Prentice-Hall, 1976).
10. Grover, P.S.: Essentials of Algol Programming (Allied Publishers, New Delhi, 1982).
11. Bentley, J.L.: Writing Efficient Programs (Prentice-Hall, 1982).
12. Shooman, M.L.: Software Engineering (McGraw-Hill, 1983).
13. Chivers, I.D.: A Practical Introduction to Standard Pascal (Wiley, 1986).
14. Rajaraman, V : Computer Programming in Pascal (PHI, 1982).

Index

Bold words are Pascal reserved identifiers

PASCAL Programming Fundamentals provides an introduction to the Pascal language, a modern computer programming language, which has been implemented on all computers—micro, mini, midi and mainframe computer systems. Pascal is being widely used to teach modern programming techniques and style. It has been accepted as one of the most important ingredients of all the Computer Science Courses.

The present book covers the Standard Pascal language. The statements and programming concepts have been explained in a simple way. There are a large number of worked examples and illustrations in the book. All the examples have been tested and computer run. Exercises having objective type and review questions and program development assignments at the end of each chapter can serve the reader to examine his understanding and design of his own programs.

This book will be of interest to students, teachers, programmers and anybody else who wants to know about Pascal, use its features to understand the program development process and the current techniques of programming.

Dr. P.S. Grover is a well-known author of books on computers and programming languages. He is a highly successful teacher and popular writer as well. At present, he is Professor in the Department of Computer Science, University of Delhi, Delhi, and also Head of the Department. He has been among the pioneers of computer education and training in the country and has contributed academically and professionally to the development and advancement of computer science as a member of many education committees and forums.

Professor Grover has published many research papers in national/international journals and supervised research students and projects.

Rs. 165.00

ALLIED PUBLISHERS LIMITED

NEW DELHI MUMBAI KOLKATA CHENNAI NAGPUR
AHMEDABAD BANGALORE HYDERABAD LUCKNOW