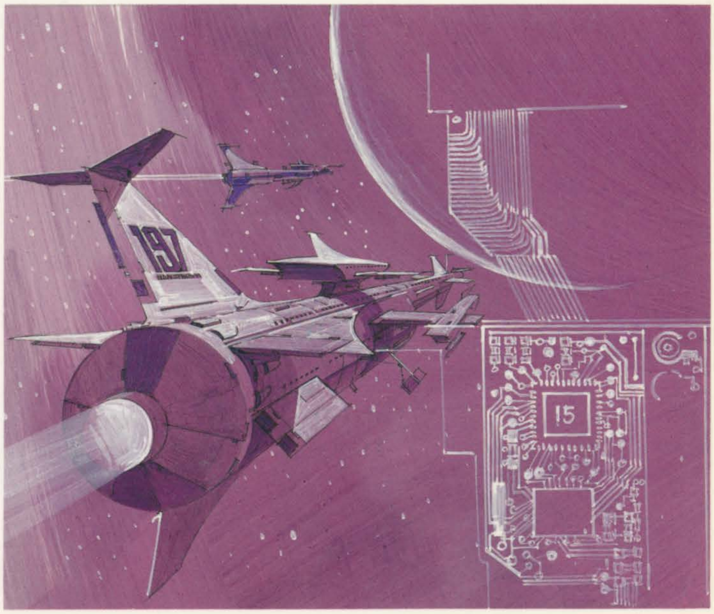
Thierry

Personal Computer

MZ-80B

PASCAL LANGUAGE MANUAL



SHARP

SHARP

Personal Computer

MZ-80B

PASCAL Language Manual

August 1981

080271-010881

Printed in Japan

NOTICE

This manual is applicable to the SB-4515 PASCAL interpreter used with the SHARP MZ-80B Personal Computer. The MZ-80B general-purpose personal computer is supported by system software which is filed in software packs (cassette tapes or diskettes).

All system software is subject to revision without prior notice, therefore, you are requested to pay special attention to file version numbers.

This manual has been carefully prepared and checked for completeness, accuracy and clarity. However, in the event that you should notice any errors or ambiguities, please feel free to contact your local Sharp representative for clarification.

Begin

This manual describes the PASCAL programming language supported by the PASCAL interpreter SB-4515. Read this manual thoroughly before using PASCAL.

The PASCAL interpreter, SB-4515, is supplied in the form of a cassette tape file.

The PASCAL language has a structure which is completely different from that of the BASIC language.
Understanding and familiarizing yourself with PASCAL programming will cause you to change your idea of programming in other languages as well.

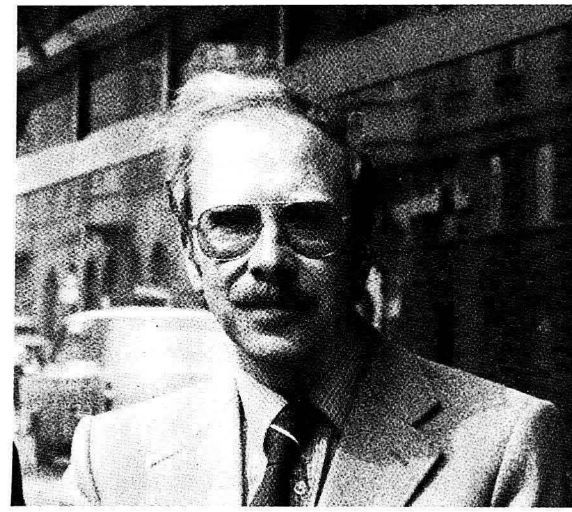
Study this manual step by step, and the sophisticated programming technique of PASCAL will be yours.

Contents

Chapter 1

Introduction

The story of PASCAL



Professor Wirth

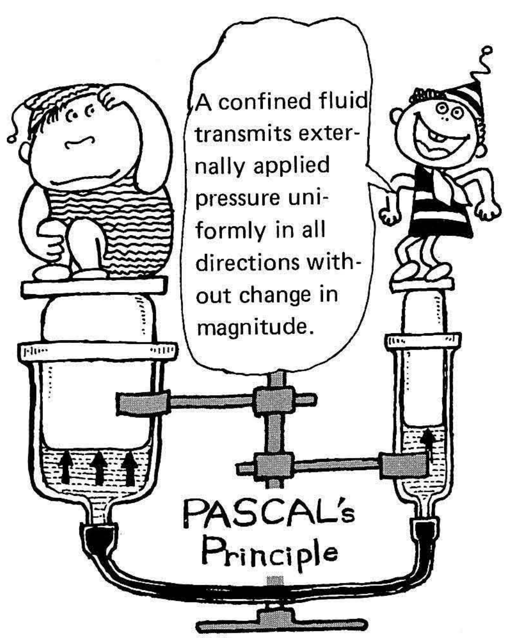
The PASCAL programming language was invented in 1968 by Professor Niklaus Wirth of Zürich. Wirth is the inventor of not only PASCAL but also of other computer programming languages.

The background for PASCAL's invention is a programming language called ALGOL 60. ALGOL 60 uses Backus notation to express algorithms in a clear and simple manner. The syntax diagrams shown from page 24 on are based on the Backus notation concept used in ALGOL 60. Although it is necessary to master PASCAL to understand the syntax diagrams, these diagrams will not appear difficult after this book has been thoroughly read.

PASCAL is named after Blaise Pascal (1623~1662), a French mathematician and philosopher who is famous as the discoverer of Pascal's principle, which he proved when he was only 16 years old, and as the inventor of a practical calculator. Professor Wirth invented PASCAL to provide a new systematic, scientific programming technique which does not require reliance upon intuition.

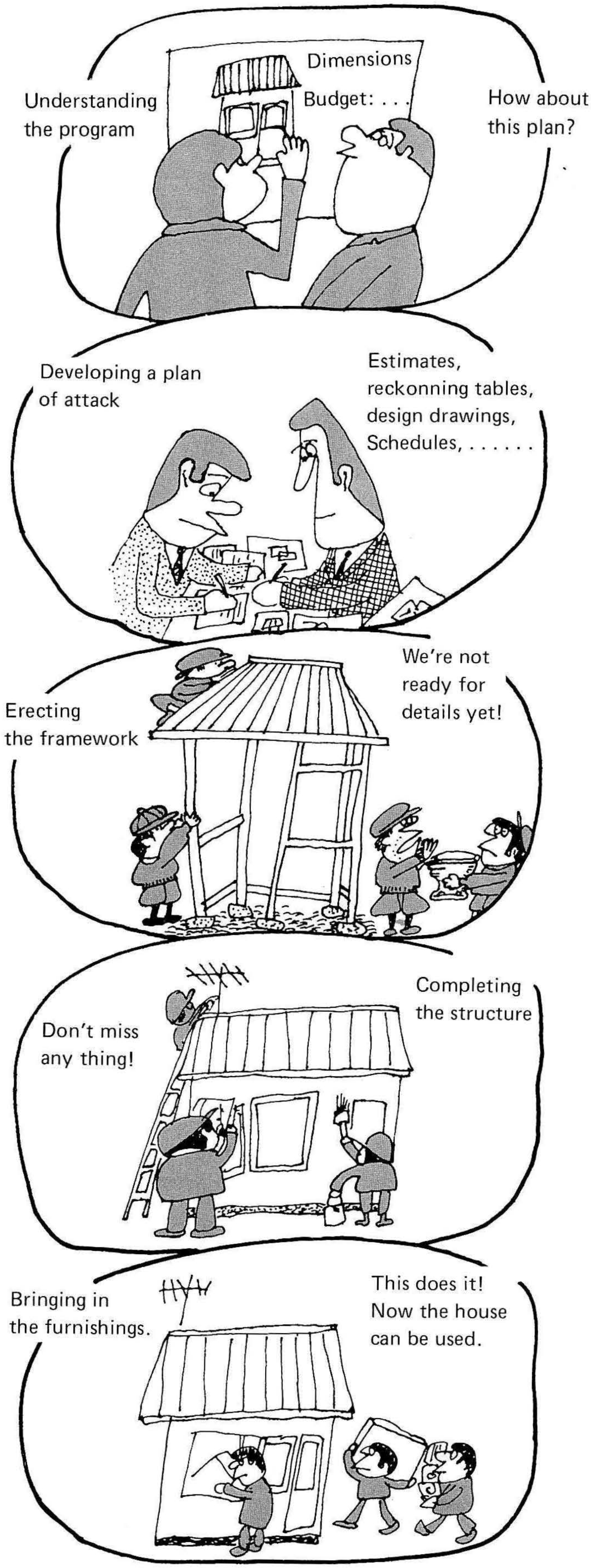
This idea did not occur to him by chance but was one of the inevitabilities of history. ALGOL 60 was established by the International Federation for Information Processing (IFIP) in 1960. Its ability to express algorithms is superior to that of FORTRAN, because of the use of Backus notation, but its input and output functions are not standardized; therefore, programs written in ALGOL 60 are not executable on different types of computers.

In 1965, ALGOL 60 was reexamined and many proposals were made for revising it. Among them was one submitted by Professor called ALGOL-W; this language is currently used by some computer systems. After much discussion, ALGOL 68 established for use around the world, however, Professor Wirth continued the studies which led to ALGOL-W and published PASCAL in 1971.



Programming is creative constructive work and careful thinking is necessary for clear understanding of the programming process. To achieve this, the following steps should be taken: first, develop a clear understanding of the nature of the problem to be solved. Next, outline the steps required for its solution. Finally, develop the details of each step. Programming in this manner is called structured programming and PASCAL makes it easy.

Structured programming is similar to building a house, as shown at right. If you think that structured programming is easy after you look at these drawings, you will master soon this elegant programming technique.



What is the difference between PASCAL and BASIC?

Let us consider a simple problem, "read two integers and print that which is larger." BASIC and PASCAL program solutions to this problem are shown below. You may think that PASCAL is difficult, since the PASCAL program uses more lines and characters than does the BASIC program. However, if a more complicated problem is solved with the two languages, it will become clear that programming is easier with PASCAL than with BASIC. These examples merely illustrate the differences between these two programming languages.

BASIC program

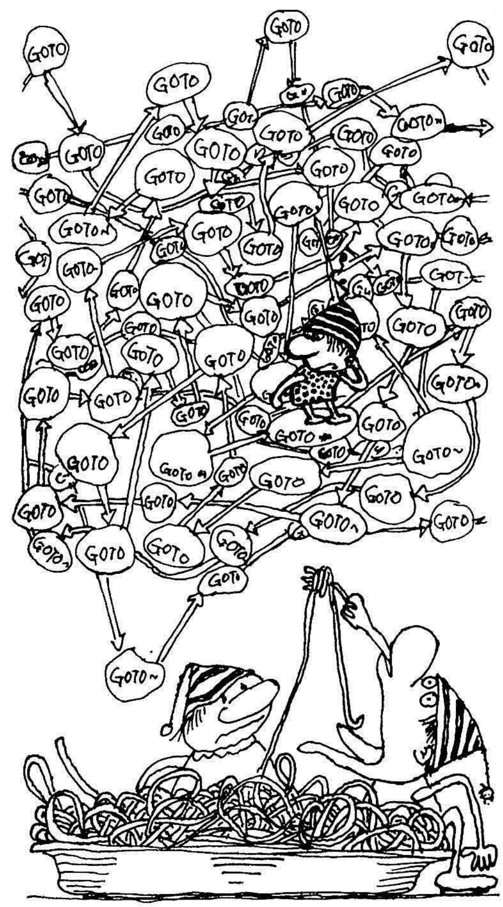
```
10 INPUT X,Y
20 IF X>Y THEN 50
30 PRINT "X < Y"
40 GOTO 60
50 PRINT "X > Y"
60 END
```

PASCAL program

```
PROCEDURE HIKAKU;
VAR X,Y: INTEGER;
BEGIN
  READLN(X,Y);
  IF X>Y THEN WRITELN("X > Y")
     ELSE WRITELN("Y > X ")
END;
```

Note that the PASCAL program contains no GOTO statement. In BASIC, it is almost impossible to write a long program without using GOTO statements. The ability to write programs without GOTO statements is a feature of PASCAL which will make itself clear as you become familiar with the PASCAL programming language.

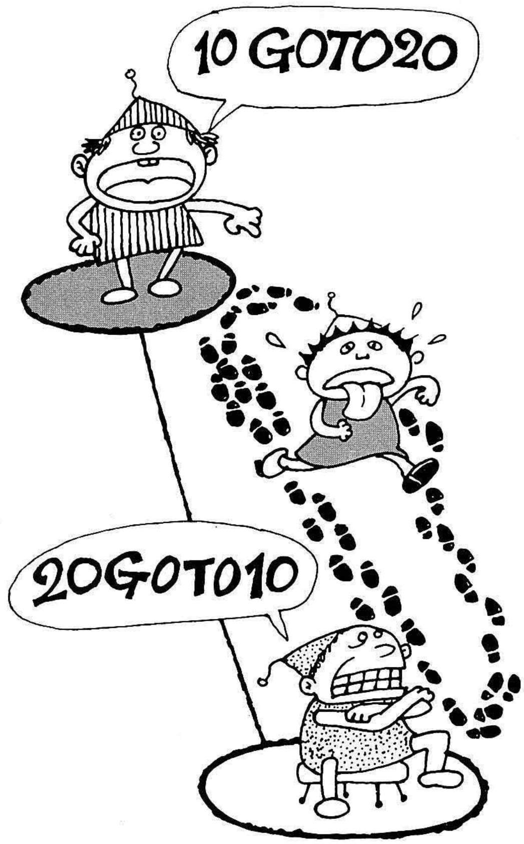
There is no real problem with GOTO statements in short programs such as the one above. They have two disagreeable characteristics, however, that tend to make them a nuisance in long programs. The first of these is that you must know the number of the program line to which execution is to move before you can finish writing the statement. This is no problem when you want to go to a section of the program which has already been written, but it can be a headache in cases where the jump is to be made to an address which is not yet known. The usual method of getting around this is to use a dummy address or a symbol in each GOTO statement, then to go back and replace them with the real addresses when the program is completed. This is not difficult when there are not many such addresses, but it can be a source of great confusion when the program is a complicated one.



The other problem becomes apparent when an attempt is made to read a program written in BASIC. Each time you come to a GOTO statement you must jump to the indicated address to see what processing is to be performed. You may have had the experience of going through seemingly endless chains of GOTO statements and despaired that you would ever be able to make heads or tails out of the mess. This type of program is sometimes referred to as a "spaghetti" program; such ill-conceived, hard to understand programs can result even when GOTO statements are used quite innocently.

GOTO is, however, a convenient statement, and it tends to be used to frequently. Since it only controls operation of the program, and does not perform any calculations or display anything on the display screen, the computer can be used most effectively by doing without it wherever possible.

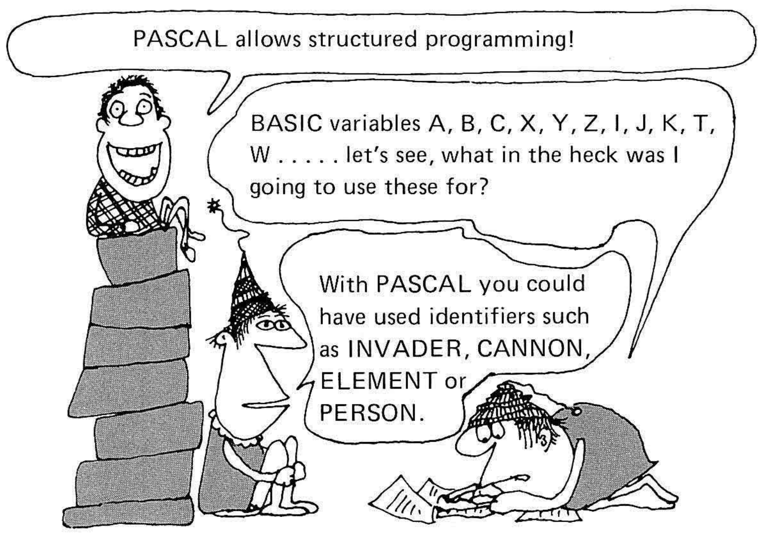
The nature of BASIC is such that the number of unneeded GOTO statements tends to increase as the length of the program grows unless the greatest care is taken in writing the program. The structured programming of PASCAL not only eliminates this problem, but reduces the likelihood that errors will occur when writing the program by encouraging an organized approach to defining the nature of problems. This makes programs easier to understand after they have been written.



Another difference between BASIC and PASCAL is in the manner in which variables are handled. Variable identifiers in BASIC are limited to a maximum of two characters, while PASCAL allows eight or more characters to be used to define a variable. The ability to use more characters in variable identifiers means that the identifiers can be more descriptive of their function in the program; for example, HOUSE instead of H, COLOR instead of C, NUMBER instead of N. HOUSE, COLOR and NUMBER all naturally convey concepts much more effectively than do the letters H, C and N. Even though little more labor may be involved in keying in such identifiers, it should be obvious that this is more than made up for by doing away with the need to have to try to remember which letter goes with which variable.

These facts do not mean that PASCAL can be used to make wonderful programs without effort; the skills involved in structured programming involve more than just familiarity with the programming language. Structured programs can also be written in BASIC, (even using the GOTO statement), as long as a well organized approach is taken in developing solutions. In fact, the effectiveness of any programming language approach taken by the user.

PASCAL makes structured programming easy. Using it leads to a natural understanding of this concept; however, it is still possible to wind up with a tangled, difficult to understand mess if care is not taken. This can best be avoided by obtaining a clear understanding of PASCAL's underlying principles.



Let's try structured programming

Let's become a little more familiar with the concept of structured thinking.

Consider the case of stereo equipment; broadly speaking, there are two basic types of such equipment: component units and music systems. As you know, in a component system, the tuner and amplifier are separate. In the more sophisticated devices the preamplifier and the amplifier are also separated. In other words, the functions which comprise the stereo system are designed as separate units, which are then combined to suit the listening taste of the user.

The component approach in stereo systems is a form of structured thinking. First, a clear understanding is developed as to the overall functions and specifications required, then each of these functions is handled as a module. Modular construction and modular furniture is based on the same concept.

The building block system used in the manufacture of construction equipment is also based on structured thinking. The overall functions of the equipment are broken down into appropriate parts (units or blocks), each of which is then designed with measurements and characteristics which will allow it to be combined with the others to obtain the desired result.

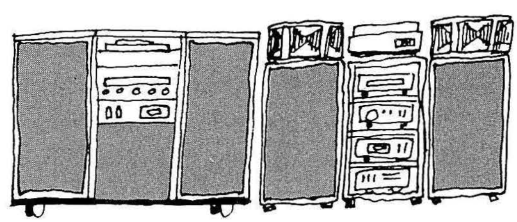
What all of these have in common is that the first step involves defining an objective and then identifying the functions, patterns or sequences which are involved in attaining it. The point is that the process starts with the overall situation, and then proceeds from top to bottom or from the outside in as details to the final solution are developed in stages.

Let's try solving the following problem as an exercise in structured thinking. As none of the PASCAL instructions have been explained yet, just follow the flow of thought.

Read in N Constants, Arrange Them in Ascending Order and Display Them.

Note that this problem can be broadly divided into three blocks. This first step can be set forth as follows.

Music system **Component units**

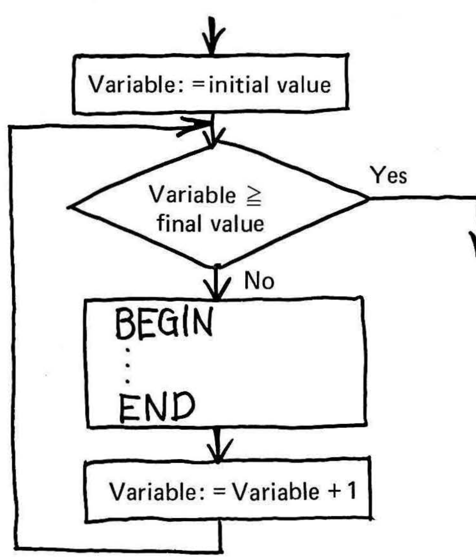


Step 1

1 N elements of data must be read in. An array must be prepared for accepting them in order of entry.
2 The data read in must be arranged in ascending order.
3 The data must be displayed after it is arranged in ascending order.

Step 1 shows the first stage of the approach which might be taken in PASCAL. The variables are not yet defined as the precise need for them has not yet been determined.

The **for** statement of PASCAL is introduced in step 2. This statement has the meaning indicated in the flowchart at right.

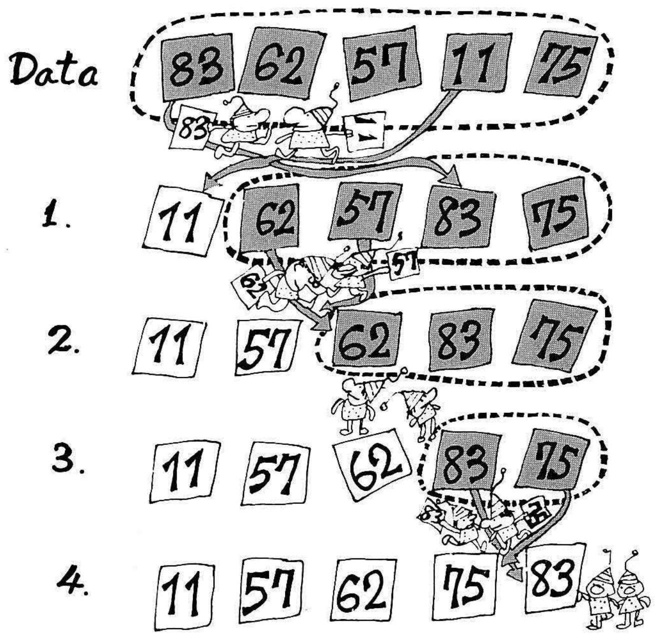



```
for Variable: = initial value to final value do
    begin
      .
      .
      .
    end ;
```

One possible procedure for arranging the values in order is as follows.

1 Search for the smallest value X[M] in the elements included in array X.
2 Exchange the value in X[1] with the value in X[M].
3 Search for the smallest value X[M] in the elements from X[2] to X[N].
4 Exchange X[2] with X[M].
5 Repeat the sequence in steps 3 and 4 for elements X[3] to X[N] until the last element is reached.

This procedure constitutes the core of step 3 in solving the problem.



The last problem remaining is that of locating the smallest value X[M]. A method for accomplishing this can be summarized as follows.

1 Assign the value in X[I] (the first element of the array) to smallest value variable MINIMUM. Assign the identifier of the first element to variable M.
2 Establish a new variable for looping, J, and repeat the following for J for I+1 through N.
- IF X[J] is smaller than MINIMUM,
  - Assign the value in X[J] to MINIMUM;
  - Assign the current value of loop variable J to M.
- If X[J] is larger than MINIMUM, go on to the next data element for comparison.

Step 4 consists of applying this procedure in a program. As can be seen, the general procedure is to first develop an overall grasp of the program, then to develop details of the solution in stages. Although the number of steps involved will vary according to the problem, the important point is that this approach provides a clearer and more certain solution than can be attained intuitively.

Step-1

PREPARES AN ARRAY OF 10 ELEMENTS THE ACTUAL NUMBER OF ELEMENS REQUIRED DEPENDS ON THE VALUE OF N.

```
VAR X:ARRAY[10] OF INTEGER;
    □,□,□,□,□:INTEGER;
BEGIN
  [READ IN DATA]
  [ARRANGE DATA IN ASCENDING ORDER]
  [DISPLAY THE DATA]
END.
```

Step-2

```
VAR X:ARRAY[10] OF INTEGER;
    N,I,□,□,□:INTEGER;
BEGIN
  READ(N);  <= READ IN THE NUMBER OF ITEM OF DATA TO BE PROCESSED
  FOR I:=1 TO N DO READ(X[I]);
  [ARRANGE THE DATA IN ASCENDING ORDER]
  FOR I:=1 TO N DO WRITE(X[I]);
END.
```

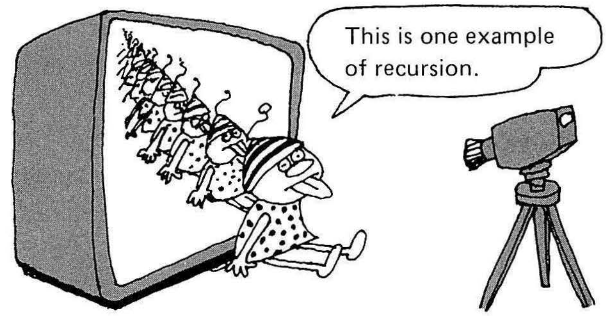
Step-3

```
VAR X:ARRAY[10] OF INTEGER;
    M,N,I,□□:INTEGER;
BEGIN
  READ(N);
  FOR I:=1 TO N DO READ(X[I]);
  FOR I:=1 TO N-1 DO
    BEGIN
      [SEARCH X[I+1] TO X[N] FOR THE SMALLEST VALUE X[M].]
      [EXCHANGE X[I] WITH X[M]]
    END;
  FOR I:=1 TO N DO WRITE(X[I])
END.
```

Step-4

```
VAR X:ARRAY[10] OF INTEGER;
    M,N,MINIMUM,I,J:INTEGER;
BEGIN
  READ(N);
  FOR I:=1 TO N DO READ(X[I]);
  FOR I:=1 TO N-1 DO
    BEGIN
      MINIMUM:=X[I];M:=I;
        FOR J:=I+1 TO N DO
          IF X[J]<MINIMUM THEN
            BEGIN
              MINIMUM:=X[J];M:=J
            END;
      X[M]:=X[I];X[I]:=MINIMUM
    END;
  FOR I:=1 TO N DO WRITE(X[I])
END.
```

Recursion:A phenomenon which can be seen in everyday life



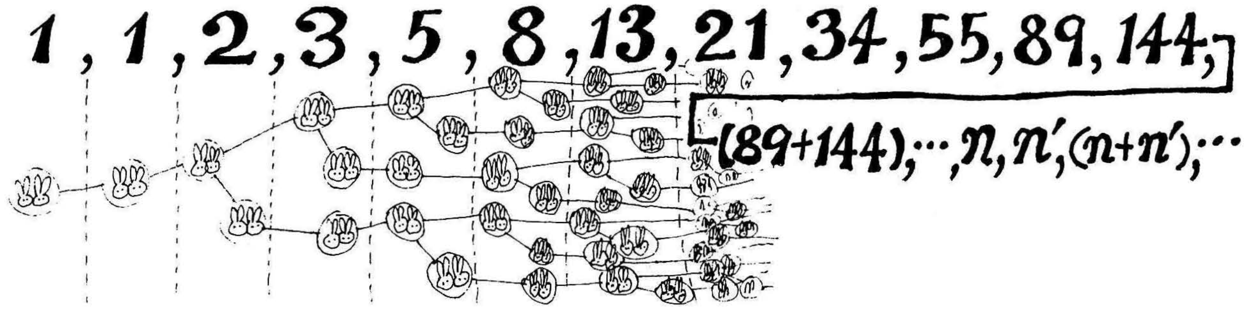
Recursion is an idea which is frequently used in PASCAL programs. We can see this phenomenon in everyday life; for example, if you sit in front of a television set with a camera which is connected to the set pointed at yourself, you would see an image something like that shown in the drawing at right. In other words, recursion is what happens when something includes itself as a part.

Let's take a look at a more concrete example.

In the 13th century, an Italian named Fibonacci conceived a mathematical sequence which he stated as follows.

"One pair of rabbits bears a litter of two pups every month, and each pair of pups starts to bear its own litters of two pups each month after one month."

A mathematical sequence which increases according to this rule is called a Fibonacci sequence. This sequence is an example of recursion because the total number of rabbits in each month is the sum of the number of rabbits in the two preceding months.



Many occurrences of this sequence can be found in the natural world. For example, careful examination of a pine cone will reveal that the scales are arranged in two types of spirals, one which winds to the left and one which winds to the right. The seeds are located at the intersections of the spirals, and the number of spirals is 5 and 8. The seeds of pineapples are located at the intersections of 8 and 13 spirals, those of English daisies at the intersections of 21 and 34 spirals and those of sunflowers at the intersections of 55 and 89 spirals.

The Fibonacci sequence is formally defined as follows:

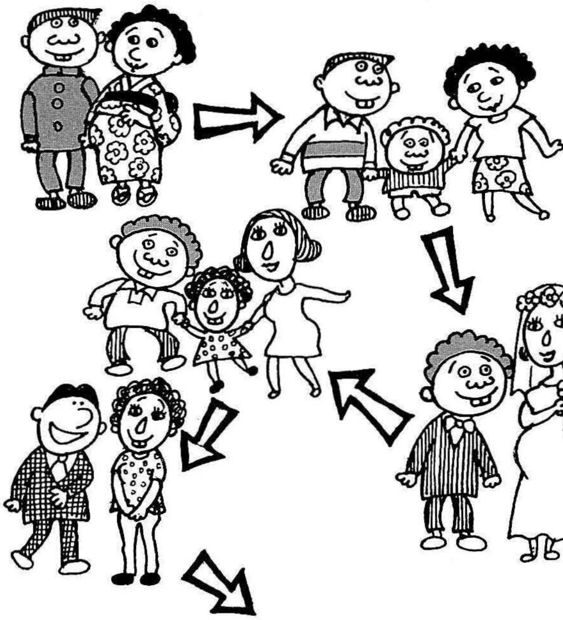
$$F(X) = \begin{cases} 0 & \text{for } X=0 \\ 1 & \text{for } X=1 \\ F(X-1) + F(X-2) & \text{for } X>1 \end{cases}$$

This can be expressed in PASCAL as shown below. As you will notice, the structure is such that an **if** statement is included within another **if** statement. Of course, recursion may be used not only with instructions, but when a part of a procedure or a function is called as is with different conditions and variables to perform an identical operation.

The structure of recursions appearing in PASCAL programs is expanded into expressions as appropriate according to their type.

```
function  F (X : integer) : integer ;
  begin  if  X=0  then  F : =0
             else  if  X=1  then  F : =1.
                   else  F : =F (X-1) +F (X-2)
  end ;
```

Here are some more examples of recursion so that you can become more familiar with this concept.



○ Method for finding the factorial n!

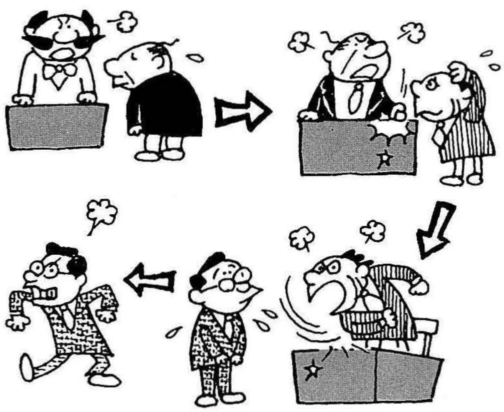
0 ! = 1
1 ! = 1 × 0 !
2 ! = 2 × 1 !
3 ! = 3 × 2 !
⋮
n ! = n × (n−1) !

○ Method for finding the total of all integers to N

1+2+3 . . . . . +N

(1+2+3+ . . . . . +N−1)+N

((1+2+3 . . . . . +N−2)+N −1)+N



○ Backus notation

⟨ number ⟩ : : = ⟨ digit ⟩ | ⟨ number ⟩ ⟨ digit ⟩
⟨ digit ⟩ : : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨ number ⟩ and ⟨ digit ⟩ each indicates a concept. The symbol : : = is used to show that the concept on the left is defined on the right. The vertical bar | is used to indicate the concept 'or'.

Therefore, ⟨ digit ⟩ indicates one of the figures from 0 to 9, and ⟨ number ⟩ indicates either a ⟨ digit ⟩ or a ⟨ number ⟩ followed by ⟨ digits ⟩. Recursion occurs because the definition for ⟨ number ⟩ includes ⟨ number ⟩ in its second half.

Since this definition indicates that ⟨ numbers ⟩ may consist of just ⟨ digits ⟩, the figures

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

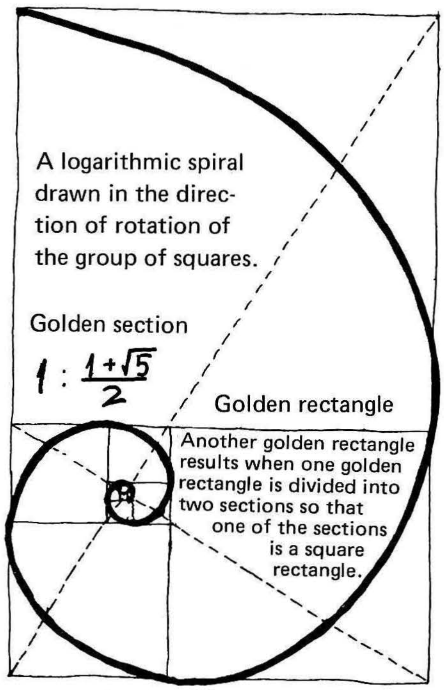
are ⟨ numbers ⟩. Since ⟨ numbers ⟩ can also consist of any of the above followed by ⟨ digits ⟩,

012, 3333, 110 9876543210

are also ⟨ numbers ⟩.

○ Logarithmic spiral of the golden section

A golden rectangle is a rectangle with dimensions such that, when it is divided with a line to form a square at one of its ends, the rectangular section which is left over has dimensions of the same relative proportions as the original. When this process is repeated many times in a fixed direction, a spiral is described which does not change its shape no matter how large or small it becomes. (This spiral is a logarithmic spiral which is drawn in the direction in which the rectangle is subdivided.) The shells of mollusks such as the nautilus have this form.



Event which do not constitute recursion

○ Simulated recursion in a BASIC program

```
10  N=15
20  PRINT N
30  IF N=0 THEN RETURN
40  N=N-1
50  GOSUB 20
60  RETURN
```

The program example above does not constitute full recursion. The reason is that the loop is repeated without the variable actually being reproduced. It is difficult to produce a program which includes full recursion, but it is easy to simulate this process. This is done by storing the initial value of the variable used in the subroutine in an array before recursion begins, then restoring the original value to the variable before returning from recursion.

A program such as the one shown below is possible if it is assumed that X is the only variable used in the subroutine. However, this still does not constitute a true example of recursion.

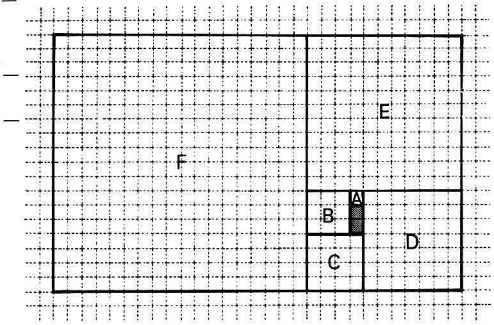
```
10  DIM X(15)
20  N=15 : X=0
30  P=0  <- Initial value of the pointer which indicates the depth of
            recursion is set to 0.
40  X=X+1 : PRINT X
50  IF N=0 THEN 120
60  N=N-1
70  GOSUB 200
80  GOSUB 40
90  GOSUB 300
100 PRINT X
110 IF P=0 THEN STOP
120 RETURN
200 REM  SAVE THE VARIABLE VALUE
210 X(P)=X
220 P=P+1
230 RETURN
300 REM  RESTORE THE VARIABLE VALUE
310 P=P-1
320 X=X(P)
330 RETURN
```

Use of recursion in this manner can best be understood by looking at it as a case in which a jump is made to a copy which is produced when the recursive image is called. Of course, if recursion also occurs in the copy, another copy is produced to which another jump is made.

Trying to visual this process mentally can be disturbing, since it is easy to infer that duplication of images can occur infinitely.

Therefore, let us emphasize that recursion is not a type of infinite loop. A recursive expression is one which defines an unlimited process within a limited description. The important point is that the limitation must be defined in the description; in actual use, the depth of recursion must be limited so that infinite repetition is avoided.

Let's take another look at the examples to highlight processes which do not constitute recursion.



Draw two different size squares adjacent to each other, such as **A** and **B**. Then draw another square, **C**, which has sides whose lengths are equal to the total of the sides of sides **A** and **B**. Next draw square **D** so that its sides are equal to the total of the sides of B and C . . . . . .
Repeating this process results in a rectangle whose dimensions approach those of a golden rectangle as more and more squares are added; that it, the ratio of its width to its height approaches ϑ, where $\vartheta = \frac{1+\sqrt{5}}{2}$.

○ Procedure for finding factorial n! :

$\infty! = \infty \times (\infty - 1)!$ does not constitute recursion, since n must be a finite ⟨ number ⟩. The important point here is that the definition of n! includes the factorial of a number $(n-1)!$ which is one smaller than n. In other words, the value of $(n-1)!$ consists of two elements until the end of the sequence is reached at 0!, which does not need to be defined as a factorial expession.

○ Backus notation:

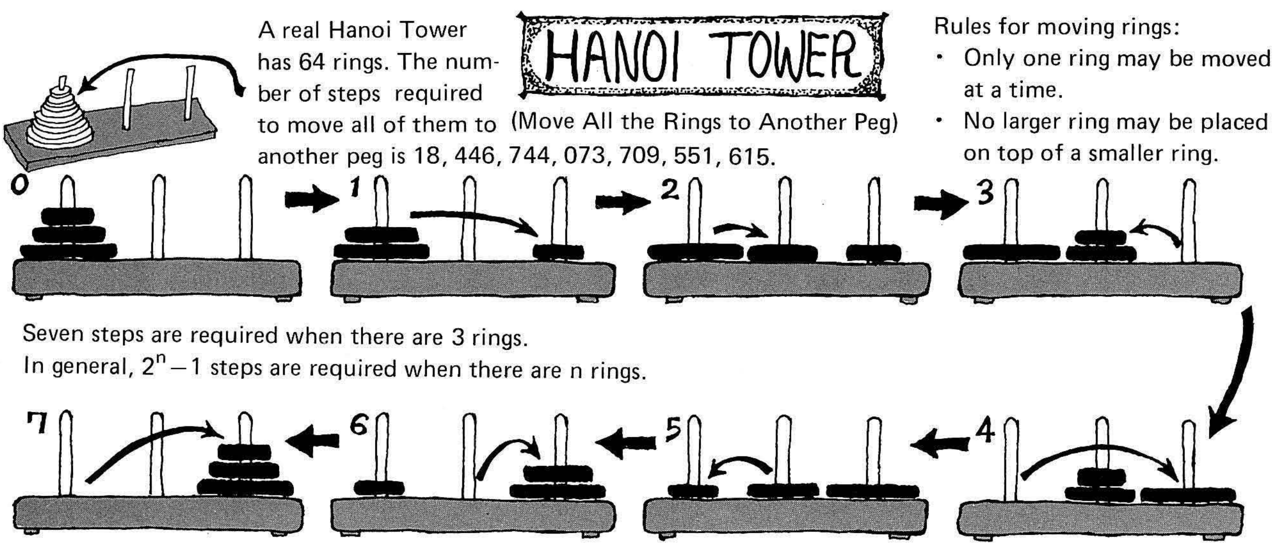
The figures 12341234. . . . . 1234 fall within the definition of ⟨ number ⟩ even if the sequence is repeated a thousand times. However, an infinite string of 9's does not constitute a ⟨ number ⟩, nor does π, since both of these continue without limit.

○ BASIC program simulation of recursion:

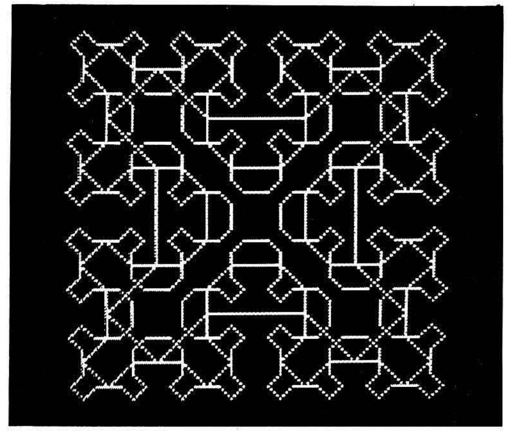
Even though the values of the variable are stacked in an array to simulate recursion, statements such as that on line 40

```
40 N=N+1
```

do not occur in true recursion.



Recursive figures



Sierpinski curves of S (1) ~ S (3)

This section introduces Sierpinski curves, beautiful patterns which are defined recursively. The illustration at right shows three different levels sizes of these patterns drawn overlying one another.

Let's reduce this pattern to its basic form to learn how it is recursively structured.

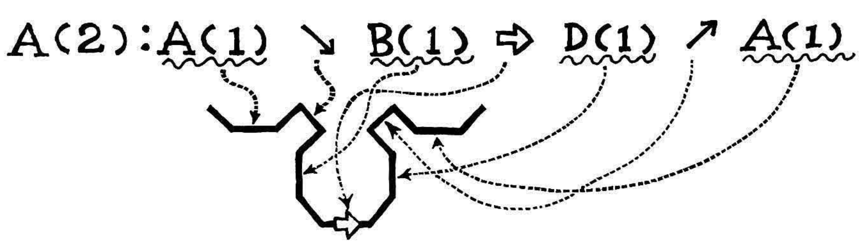
A Sierpinski curve of size n is defined with the following statement:

S (n) : A (n) ↘ B (n) ↙ C (n) ↖ D (n) ↗

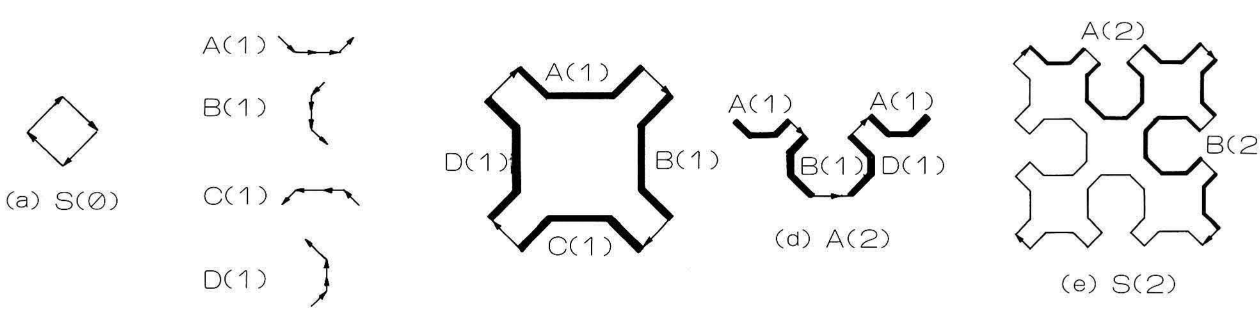
A (n), B (n), C (n) and D (n) express steps used to draw an n size Sierpinski curve. In other words, ↘ indicates that a pattern segment is to be drawn downward at a 45° angle to the right. Thus, S (0) would be displayed in the sequence ↘ ↙ ↖ ↗, resulting in a tilted box as shown in Figure (a). A recursive pattern can be generated as shown below by expanding this process using A (n) ~ D (n) for the recursive definition. (The bold face arrows indicate segments whose lengths are twice that of other segments.)

A (n) : A (n−1) ↘ B (n−1) ➡ D (n−1) ↗ A (n−1)
B (n) : B (n−1) ↙ C (n−1) ⬇ A (n−1) ↘ B (n−1)
C (n) : C (n−1) ↖ D (n−1) ⬅ B (n−1) ↙ C (n−1)
D (n) : D (n−1) ↗ A (n−1) ⬆ C (n−1) ↖ D (n−1)

This may appear confusing at first, but A (1), B (1), C (1) and D (1) simply indicate basic patterns as shown in Figure (b). Curve S (1) is obtained as shown in Figure (c) as a natural result of the manner in which A (1) ~ D (1). A (2) is also defined using A (1) ~ D (1).



Thus, pattern S (2) is reproduced recursively by using A (2) ~ D (2) as shown in Figure (e). Increasing n by 1 and halving the length of the basic pattern, makes it possible to display multiple levels of the pattern on top of each other. This is the procedure which was used to draw S (1) through S (3), the three overlying patterns shown in the illustration at the top of this page.



(a) S(0)

(b) A(1)~D(1)

(c) S(1)

(d) A(2)

(e) S(2)

Chapter 2

Editing

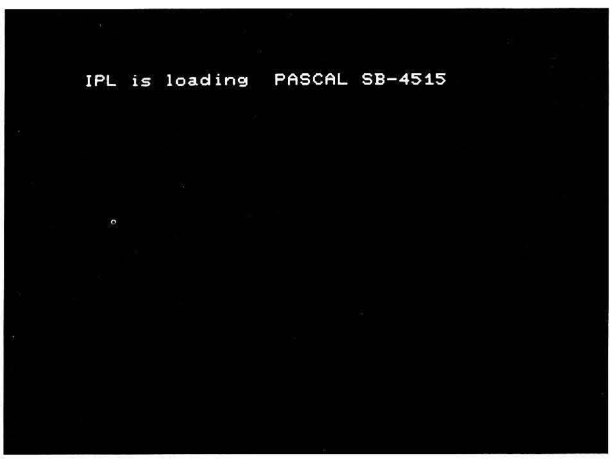
Editing is the process of creating or modifying a program, or of inserting or deleting characters in a program.

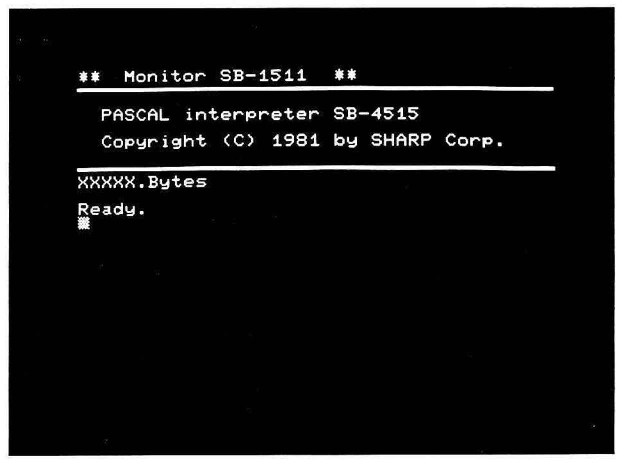
Operating the computer

Seeing is believing! Let's start our adventure in the world of PASCAL by going over procedures for operating the computer (the MZ-80B) under control of PASCAL interpreter SB-4515 series and loading, storing and modifying PASCAL programs. PASCAL syntax will be explained in the next chapter.

PASCAL SB-4515 is stored (along with Monitor SB-1511) on a cassette tape file in the same manner as the BASIC interpreter, and must undergo initial program loading whenever it is to be used. Simply place the PASCAL cassette file in the cassette tape deck and turn on the power, the IPL automatically loads both the PASCAL interpreter SB-4515 and the Monitor SB-1511 (photo at left). Upon completion of loading, the MZ-80B displays the message illustrated in the photo at right and the PASCAL interpreter being to operate.

Instead of "xxxxx", the number of unused bytes of memory in the computer may be indicated.





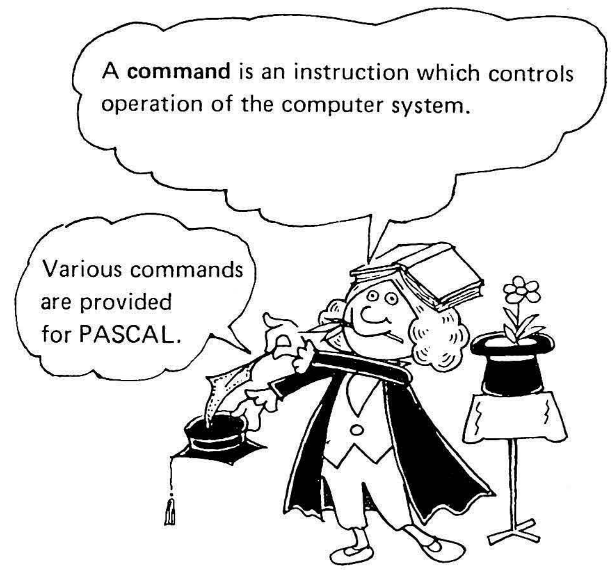
Load command A (Append)

Load the program stored in the first file of the PASCAL Application Tape. (A listing of this program is shown in Sample Programs.)

Key in A, then CR. This corresponds to the LOAD command in BASIC.

The computer then requests that the name of the file to be loaded be input by displaying "Filename?" on the CRT screen. Key in Hanoi Tower, then CR. (Alternatively, just key in CR.) The program first encountered is loaded if only CR is keyed in. It is not necessary to enclose the file name in double quotation marks.

The cursor appears again and begins to flicker after the program has been loaded. The computer is now ready to execute the program.



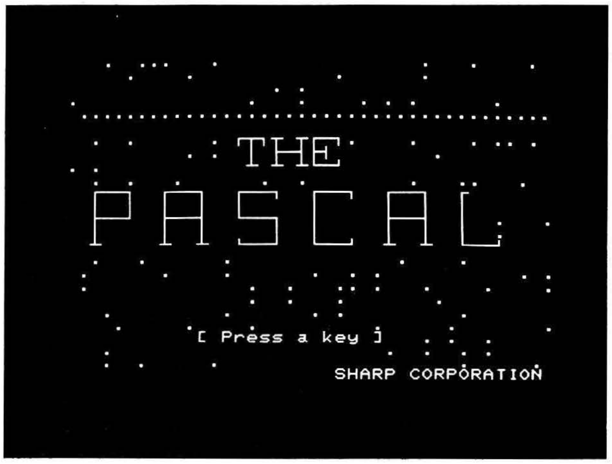
Note: If the load command is executed while a program is already stored in memory, the program loaded is stored starting at the first memory location following the existing program. This function is convenient for loading large programs.

Program execution starts with the program first loaded when the G command (G: Go) is entered.

Go command [G]

Key in [G], then [CR] to execute the program loaded. This command corresponds to the RUN command in BASIC. It takes a moment for display to begin because the computer first checks for syntax errors.

The display screen is then cleared, a starry sky appears and a message is displayed (photo at right).



Interrupting program execution: [BREAK]

Interrupt program execution by pressing the [BREAK] key.

List command [P] or [H] (for printer)

Key in [P], then [CR]. The program listing is then displayed on the CRT screen. This command corresponds to the LIST command in BASIC. Key in [H], then [CR] to print out the program listing on the printer. (The program listing is not displayed on the CRT screen in this case.) Pressing the space bar while the program listing is being output stops operation; pressing it again restarts output.

The numbers followed by periods at the left of each row of the listing are line numbers. Line numbers are always incremented in units of one.

Listing a specified line:

P line number or H line number (for printer)

Key in [P], the line number and [CR]. For example, to display the contents of line 5, key in [P], [5] and [CR]. No period, ".", is required.

Listing lines within a specified range:

P <starting line number> – <ending line number>

For example, to display lines 5 through 12, key in [P], [5], [–], [1], [2] and [CR]. To print out the lines on the printer, key in [H] <starting line number> – <ending line number>.

Listing up to a specified line:

P – <ending line number> or H – <ending line number> (for printer)

Lines are listed from the start of the program to the specified line. For example, keying in [P], [–], [2], [0], and [CR] lists all lines up to line 20.

Listing all lines from a specific line to the end of the program:

P <starting line number> – or H <starting line number> – (for printer)

Lines are listed from the specified line to the end of the program.

Listing lines without line numbers
(only applicable for the printer): #

A program listing without line numbers can be obtained on the printer by keying in [#] and [CR] before entering the list command.

You may notice that some of the program lines are indented. **Indentation** is characteristic of the manner in which PASCAL programs are written. Indentation will be explained later.

Modifying PASCAL programs

Practice the below operations and become familiar with them. This will help you to understand explanations in the following sections.

Correcting Part of a Program

Enter P, 3, 6, CR to display line 36 of the program. Replace 64 with 135 in the same manner as in BASIC. Press the CR key.

36. UFO : =CHR (64) ;

36. UFO : =CHR (135) ;

Execute P36 again to check the result. Try executing the program if the revision has been correctly made. The UFO which moves from left to right on the screen should now appear in rectangular.

When there are many lines to be corrected, it is convenient to specify listing of a range of lines. This technique is the same as that used in BASIC.

Let's try an interesting experiment at this point. Enter P, 3, 0, –, 4, 4, CR to list lines 30 through 44, then remove the indentations so that the program appears as shown at right; now try executing the program. As you can see, its operation is not affected; thus it is natural to wonder what purpose indentation serves. Although this will be explained in more detail later, briefly it serves to make the program easier to read.

```
P30-44
  30.procedure UFOMOVE;{ UFO }
  31.var N:integer;UFO:char;
  32.begin
  33.write("▣");
  34.for N:=1 to 39 do write("⇨");
  35.for N:=1 to 4 do write("⇩");
  36.UFO:=chr(135);
  37.write(UFO:1);
  38.for N:=1 to 39 do
  39.begin
  40.DELAY(100);
  41.write("⇦.⇦⇨",UFO:1)
  42.end;
  43.write("⇦ ⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩")
  44.end;
```

Delete command D Partial Program Deletion

The program delete command is used to delete one or more lines of a program.

Deletion of one specific line . . . . . . D < line number >

Try deleting line 9 of the program. Enter D, 9, CR. Line 9 is deleted and all following lines are moved up one line. List the program to confirm this.

Deleting a specific group of lines:

D < starting line number > – < ending line number >

Key in D, 5, –, 1, 0, CR. This causes the program entries on lines 5 through 10 to be deleted and all following lines of the program to be moved up six lines. In other words, program lines 11 and on are moved up to close the gap.

Deleting all lines up to a specific line number:

D – < ending line number >

Keying in D, –, 5, CR causes all lines from the beginning of the program up through line 5 to be deleted and lines 6 and on to be moved to the front of the program.

Deleting all lines after a specific line number:

D < starting line number > –

Keying in D, 8, –, CR causes all lines from 8 on to be deleted.

Note: It is possible to delete a line by enterging < line number > CR ; this method should not be used, however, since difficulty may occur depending on whether the cursor is located at the position of a character on the screen. The D command should always be used when deleting program lines.

Kill command | K/

Entering K, /, CR will cause the entire program to be erased. This corresponds to the NEW command in BASIC.

Input command | B

This command is used when a program is to be entered from the keyboard.

Inputting programs

Enter the program shown at right with the sequence described below. Be sure to enter K, /, CR before beginning.

begin
write (" © ABC")
end.

1. Key in B ↲. "0." is displayed and input of the program awaited. The symbol "↲" indicates a carriage return.
2. Enter BEGIN ↲. "1." is displayed and the next entry awaited.
3. Enter WRITE ("©ABC")↲. "2." is displayed and the next entry awaited. (© is entered by pressing [CLR HOME] and [SHIFT] at the GRPH mode.)
4. Enter END.↲. "3." is displayed and the next entry awaited.
5. Since this is the end of the program, enter only [CR]. This causes command entry to be awaited without any line numbers being displayed.
6. Confirm that the program entries are correct by entering P ↲ to execute the list command. If entries have been correctly made, execute the program by entering G ↲. The screen should be cleared and then "ABC" displayed. Try changing the characters enclosed in quotation marks and reexecuting the program.

Insert command: <line number> ▩

Let's try making an insertion in the program entered above. The insertion is to be made between lines 1 and 2.

1. 2 ▩ WRITE ("DEF") ↲ . No other entries are required. Now list the program to confirm that the entry has been made correctly. (▩ is entered by pressing [SHIFT] and [TAB] keys simultaneously.)
2. When this program is executed, the error message **∗ Err 18 ∗ Line** 2. is displayed and execution halts. The reason for this is that the entry made in step 1 results in a syntax error. Correct the program as indicated below.
3. Add a semicolon (;) to the end of *write* ("©ABC") to separate it from the next statement and execute the program; now "ABCDEF" should be displayed. The error resulted because the computer did not know where the command on line 1 ended. It does not matter whether a semicolon is included at the end of line 3 for reasons which will be explained later.

   Try making all program entries on one line and executing the program as shown below; the result should be the same.

   begin *write* (" ©ABC") ; *write* ("DEF") **end.**
4. When more than one program line is to be inserted, execute <first insertion line number>▩↲. Line numbers are displayed and program input (program input ↲) awaited; then the next line number is displayed for entry of another statement. As many program lines can be entered as necessary.
5. The input command is terminated by entering only a carriage return when the next line number if displayed.

Making an insertion at the beginning of the program: B

This command is used to make an insertion at the beginning of the program. Entering B↵ causes "0." to be displayed and entry of the insertion to be awaited. This allows a new line to be entered at the beginning of a program. The entry is terminated with CR .

Making an insertion at the end of the program: Z

This command is used to make an insertion at the end of the program. Entering Z↵ causes the line number following that of the last line of the program to be displayed. For example, when the number of the last line of the program is 35, "36." is displayed and entry of the insertion awaited. The insertion is added to the end of the program when the entry is completed by entering CR .

Assignment of line numbers is not fixed as in BASIC, but change as insertions and deletions are made. In the process of programming, you will often find that you have called up a line other than the one which you wanted to review or that you want to review the lines before and after a specific line. The L (Last) and N (Next) commands explained below are useful in such situations.

L command

L < number of lines to be reversed >

When line 10 of a program is listed by executing P10↵, line 8 can also be reviewed by entering L 2 ↵ . If L 3 ↵ is entered next, line number 5 will be listed.

When ▦↵ is entered after an L command is executed, insertions can be made in the program from the line displayed by the command. In other words, this command can be used in the same manner as the insert command.

Line 0 will be displayed if the number of lines specified in the command is greater than the number of lines in the program.

N command

N < number of lines to be advanced >

This command functions in a manner similar to the L command. If N 5 ↵ is entered after P10↵ is executed, line number 15 will be listed. An insertion can be made after line 15 by entering ▦↵. The number of the line following the last line of the program will be displayed if the number of lines specified in the command is greater than the number of lines in the program.

M command

M

The number of unused bytes of memory in the computer can be displayed by entering M↵ . This corresponds to PRINT SIZE in BASIC.

E command

E$ < hexadecimal address >

Specifies the maximum amount of memory which can be used by a program. The full memory will be available unless otherwise specified with this command. For example, if E$A000 is entered, the limit is set at address $A000. Since the address is specified in hexadecimal notation, "$" is mandatory. The specifiable range is from $8000 to $FFFF.

This corresponds to the LIMIT instruction in BASIC.

S command S

This command is used to save a program on cassette tape. It corresponds to SAVE in BASIC. Let's try this.
Enter the program shown below after executing the K/ command.

```
begin
    write ("ABC")
end.
```

Next, enter S ↵, "Filename?" will be displayed on the screen to prompt assignment of a file name, so a suitable name must be given to the file. The file name is composed of a string of up to 16 characters. If no file name is specified, the above program file will have no name and later identification will be difficult.

V command V

This command compares the program contained in the text area with its equivalent text (file name: *file name*) in the cassette tape just saved by S command. It corresponds to VERIFY in BASIC.

If the program and tape file coincide, "OK" will appear on the screen; otherwise, "Error" is displayed.

Q command Q/

Entering Q/ ↵ causes program control to be returned from the PASCAL editor to the monitor program. And wait input of a command at the Monitor SB-1511 level. This corresponds to MON in BASIC. The entries used to return control from the Monitor to the PASCAL are;

∗ J
J-adr. **$1300** . . . . . . Cold start
$1301 . . . . . . Hot start

A cold start is one made when all programs are completely cleared and the stack pointer, etc. is initialized. This is the same as the status just after loading the interpreter.

A hot start is one made when control is passed to the PASCAL interpreter without programs being cleared or the registers initialized.

I command I/

This command activates the MZ-80B System IPL (Initial Program Loader). This corresponds to BOOT in BASIC.

$ command $ (indentation command)

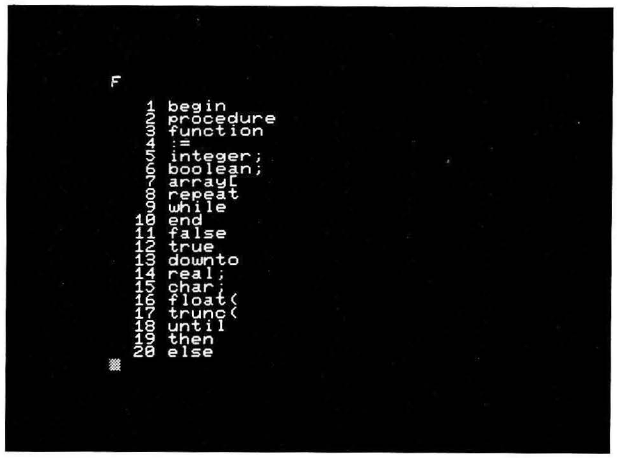
Indentation is commonly used for list representation in PASCAL programs to improve readability. The $ command, once executed, causes the editor to automatically align the current subsequent lines with the start of the preceding line.

Entering the $ command again disables the indentation mode.

F command F

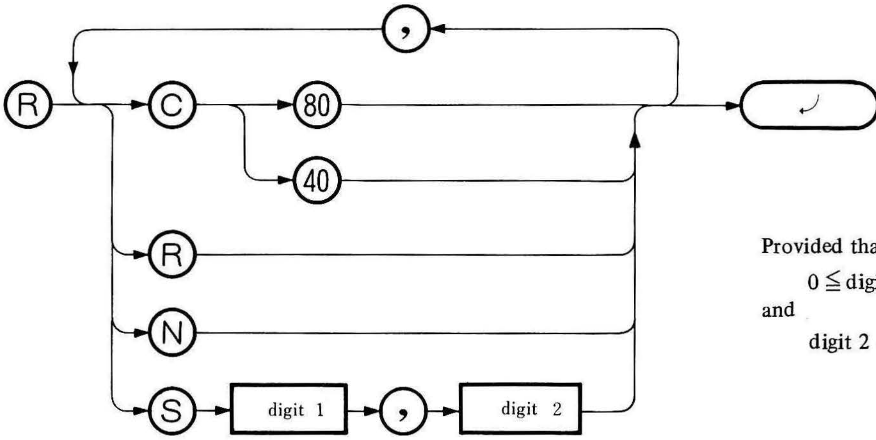
This command displays a complete list of string definitions for definable function keys, thereby enabling you to determine how individual definable function keys are defined. This corresponds to KLIST in BASIC.

The string definitions of each definable function keys are initially defined by the PASCAL interpreter as follows.



R command R

This command includes the syntax structure represented by the syntax diagram below.



Provided that

$0 \leqq$ digit 1 $<$ digit 2 $\leqq 24$

and

digit 2 $-$ digit 1 $\geqq 2$.

The operand of the R command (R: range) determines which of three functions shown below are activated.

1) Changing the character display mode

C80 . . . . Sets the character display mode to "80 characters/line".

C40 . . . . Sets the character display mode to "40 characters/line".

2) Changing the character and graphic display mode

R . . . . Sets the character and graphic display mode to reverse mode.

N . . . . Sets the character and graphic display mode to normal mode.

3) Fixing the scrolling area

S digit 1, digit 2 . . . . digit 1 and digit 2 fix the scrolling area. The top line refers to line 0 of display and the bottom line to line 24.

Editor command table

COMMAND		OPERATION
Append command	A ↵	Appends a program from the cassette tape to the program in memory.
Go command	G ↵	Executes the program.
List command (to CRT display)	P ↵ P ⟨line number⟩ ↵ P ⟨starting line number⟩ – ⟨ending line number⟩ ↵	Outputs the entire program listing. Outputs a specified line of the program listing. Outputs a specified range of lines of the program listing.
List command (to printer)	H ↵ H ⟨line number⟩ ↵ H ⟨starting line number⟩– ⟨ending line number⟩ ↵ #↵	Outputs the entire program listing. Outputs a specified line of the program listing. Outputs a specified range of lines of the program listing. Executing # once eliminates the line numbers from the output program listing. Executing it again restores the line numbers.
Delete command	D ⟨line number⟩ ↵ or ⟨line number⟩ ↵ D ⟨starting line number⟩ – ⟨ending line number⟩ ↵	Deletes a specified line of the program. Deletes a specified range of lines of the program.
Kill command	K/ ↵	Erases the entire program.
Input command	B ↵ Z ↵ ▩ ↵ ▩ ⟨statement⟩ ↵ ⟨line number⟩▩ ↵ ⟨line number⟩▩ ⟨statement⟩ ↵ $ ↵	Used to enter a program starting at line 0. If another program already exists, the new entries are inserted in front of it. Used to enter a program starting at the first unused line following an existing program. Displays the number of the line at which the pointer is located and allows insertions to be made at the indicated line. Allows entry of one program line at the line indicated by the pointer. Allows insertion of program entries starting at the specified line. Used to insert one program line at the specified line number. Enables the editor to enter a program with indentation.
Pointer shift command	L ⟨number of lines⟩ ↵ N ⟨number of lines⟩ ↵	Moves the pointer back by the specified number of lines. Advances the pointer by the specified number of lines.
Save command	S ↵	Saves the program in memory on the cassette tape.
Verify command	V ↵	Compares the program contained in the text area with its equivalent text in the cassette tape just saved by S command.
System commands	RC80 ↵ (or C40) RR ↵ (or N) RS *ls*, *le* ↵ F ↵ M ↵ E$ ⟨address⟩ ↵ Q/ ↵ I/ ↵	Sets character display mode to 80 char./line (or 40 char./line). Sets display mode to reverse mode (or normal mode). Fixes the scrolling area to line *ls* through line *le*. Displays a complete list of string definitions for function keys. Displays the amount of unused memory area in bytes. Specifies the limiting address of memory available for program use in hexadecimal. Transfers control to the monitor. Activates the MZ-80B System IPL (Initial Program Loader).

Note: ↵ indicates pressing the CR key.

Chapter 3

Basic Rules of PASCAL

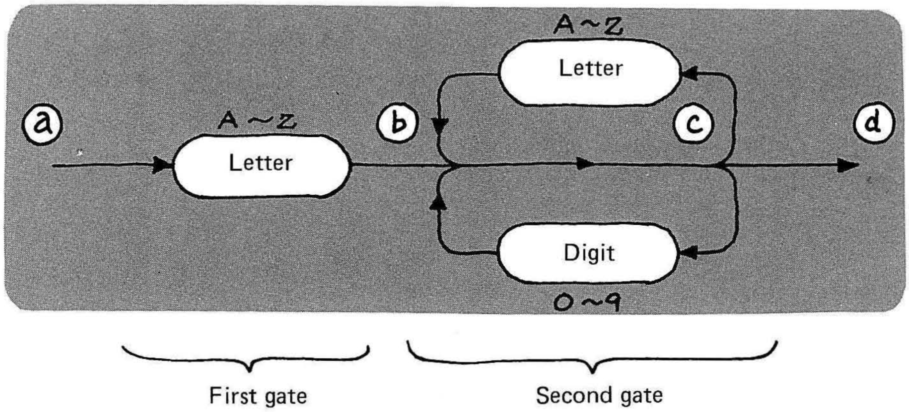
Syntax diagram

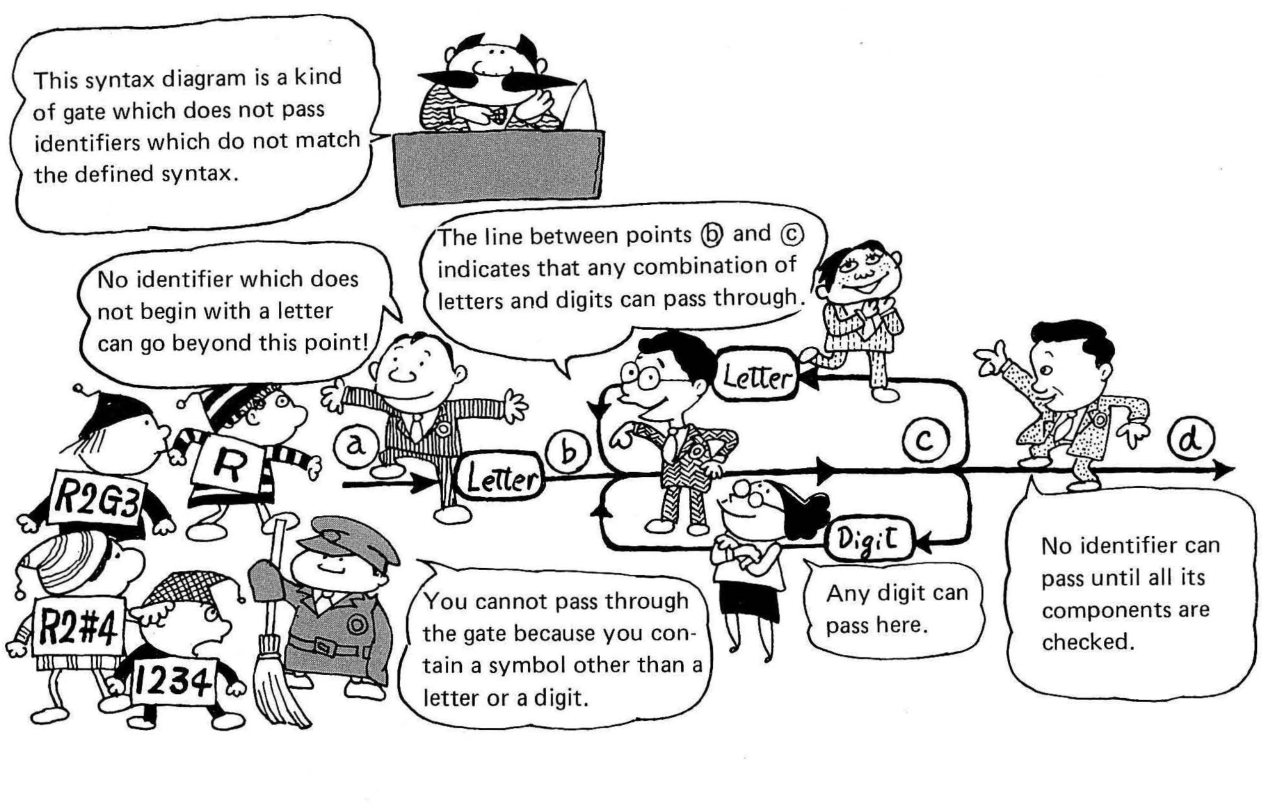
All programs must be coded according to PASCAL's own syntax. PASCAL syntax is represented by syntax diagrams, which are summarized in Chapter 7. This paragraph uses some examples to show how syntax diagrams are used.

The syntax diagrams explained in the paragraphs of this and succeeding chapters are shown at the end of each paragraph.

Example 1 Syntax Diagram for Identifiers

Various identifiers are used in a PASCAL program. For example, variables, procedures and functions are all assigned identifiers. An identifier must begin with a letter and may be followed by any combination of letters and digits. This is represented by the syntax diagram below.





Rectangular Boxes and Round-ended Boxes

Round-ended boxes enclose elements which cannot be divided in a grammatical manner. For example, (letter) represents a letter from A to Z and (digit) a digit from 0 to 9.

Rectangular boxes enclose elements which can be divided further and which are defined elsewhere. For example, [identifier] is defined elsewhere with another syntax diagram.

Look at the syntax diagram on the preceding page again.

(1) (a) is the entrance to the syntax diagram. The first letter indicates that the first element of the identifier must be a letter.

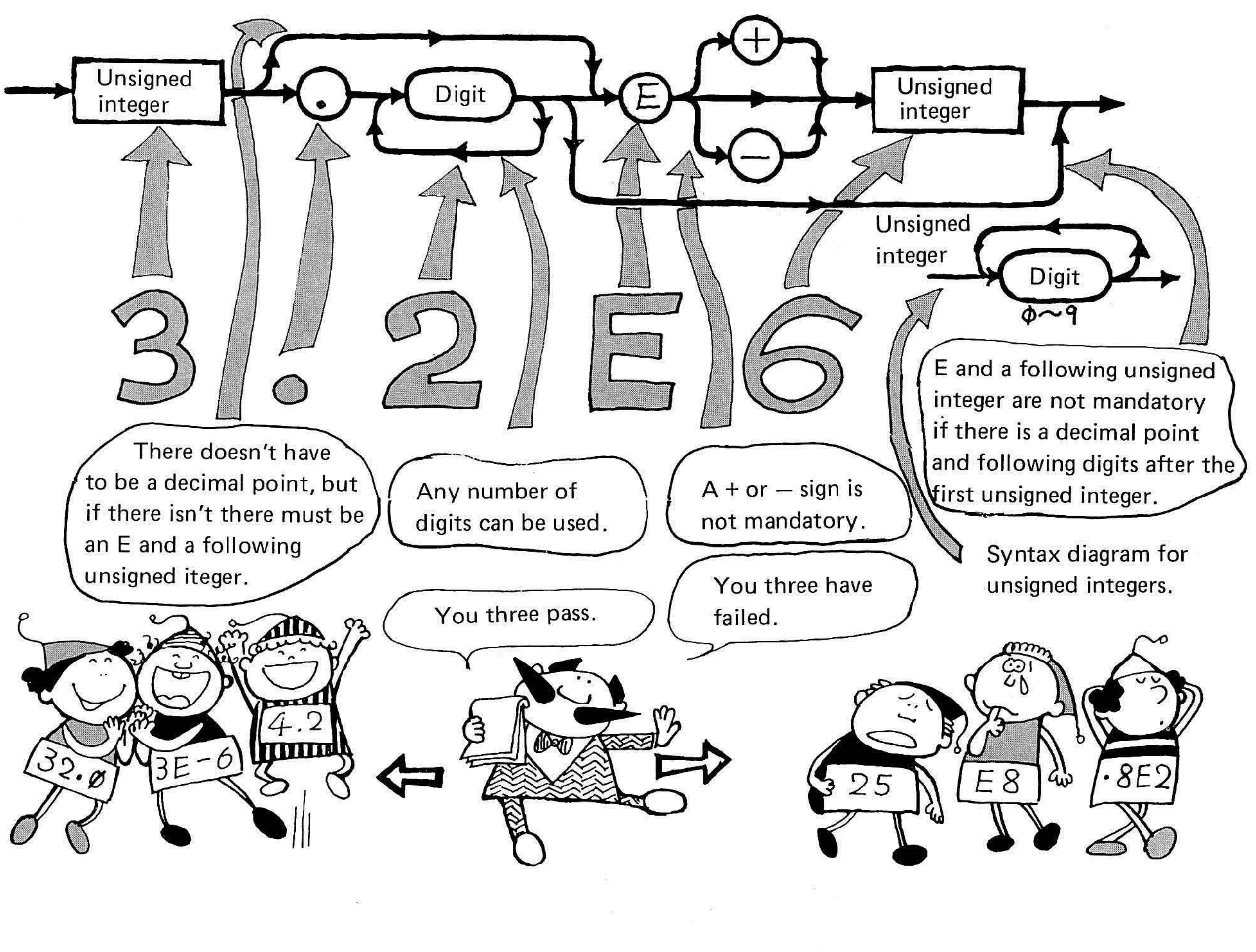
(2) The section between points (b) and (c) is the part of the syntax diagram used to check the second and succeeding elements. Identifiers can pass through this section once all their elements have been checked. Therefore, an identifier consisting just one letter can immediately pass through this section.

(3) All elements after the first letter take either the upper or lower loops, depending on whether the character being checked is a letter or a digit.

In any other case, cannot pass the syntax diagram. All identifiers are checked in the above manner, and ones which are grammatically correct pass the syntax diagram. For example, R and R2G3 are correct identifiers but R2 #4, 1234 and π are incorrect identifiers.

Example 2 Syntax Diagram for Unsigned Numbers

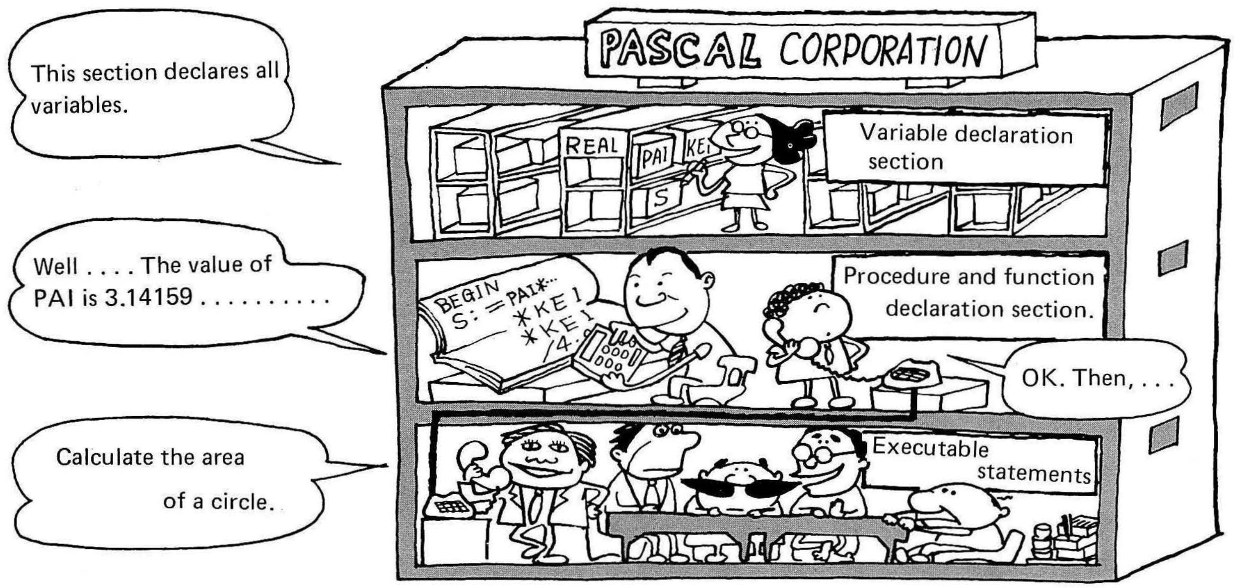
The following syntax diagram is for checking unsigned numbers. Confirm that the unsigned number 3.2E6 passes the syntax diagram. (3.2E6 represents 3.2×10^6). Does 2E2 pass?



PASCAL program structure

PASCAL programs have a certain structure which conforms to certain rules. Each PASCAL program consists of 3 sections: the variable declaration section; the procedure and function declaration section and; the executable section. These sections must be arranged in this order.

(1) Variable declaration section
(2) Procedure declaration section
(3) Executable statements



Sample Program: Computing the Area of a Circle

```
var PAI, RADIUS, AREA : real;
procedure CALCULATE (X : real) ;
    begin AREA : =PAI*X*X  end ;
begin
    PAI : =3.14159 ;
    readln (RADIUS) ;
    while RADIUS<>0.0 do
      begin
        CALCULATE (RADIUS) ;
        writeln ( " S= " , AREA) ;
        readln (RADIUS)
      end
end.
```

Variable declaration section
Procedure declaration section
Executable statements

This program reads the value of the radius of a circle from the keyboard, calculates the area of the circle and displays the result on the CRT screen. The program stops when "0" is keyed in.

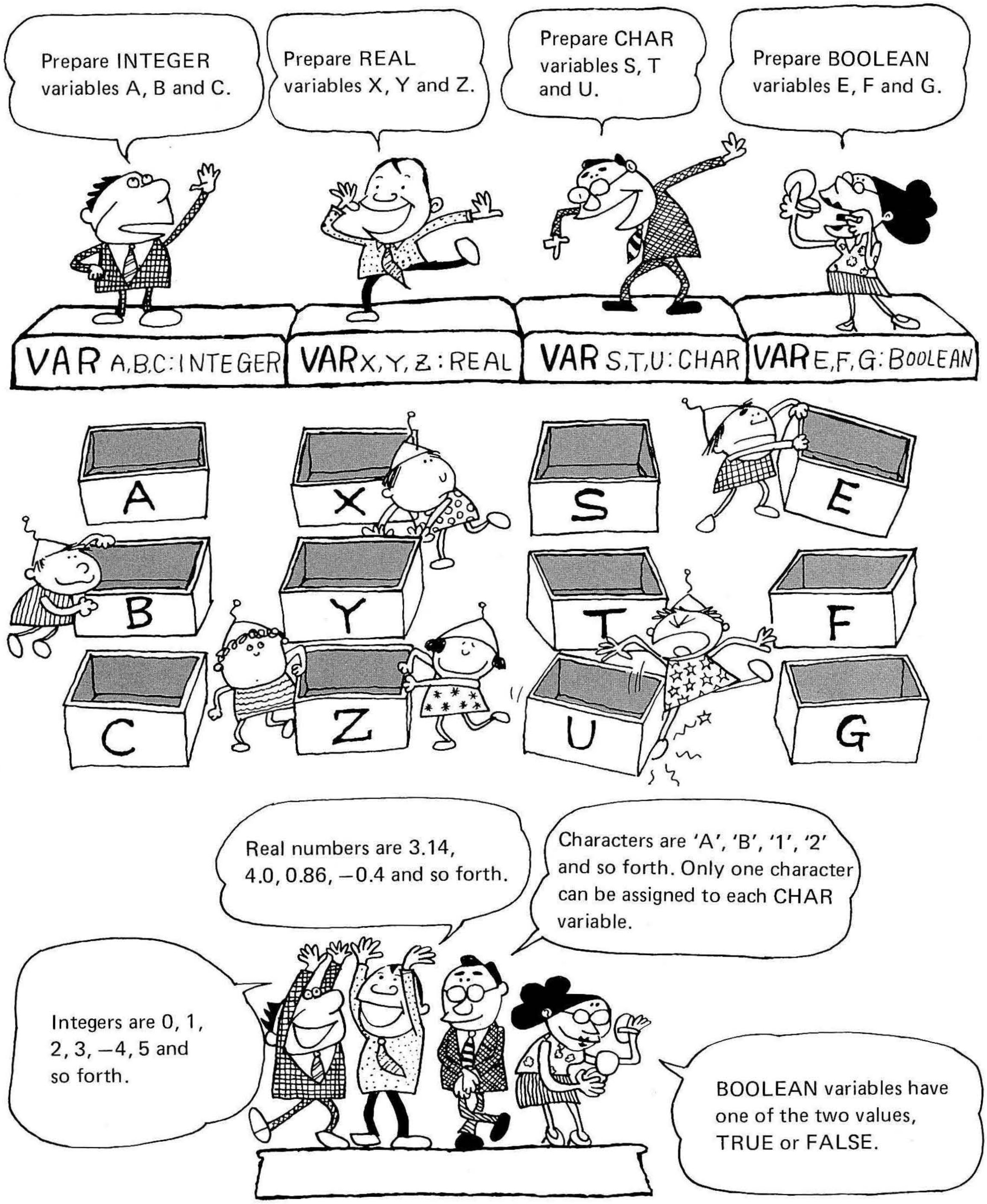
Words shown in **bold face type** are special words with fixed meanings. It is not necessary to distinguish between the two type faces when you are keying in entries.

Reserved words are listed on page 129. In PASCAL programs, integers expressed as real numbers must be followed by a decimal point and 0 (.0); for example, 3 is expressed as 3.0 and 12 is expressed as 12.0. This is not necessary for data which is read from the keyboard by the *read* statement, since it is automatically converted to the correct format by the computer. In the above program, PAI: = 3.14159 cannot be replaced with π: = 3.14159 because π cannot be used as an indentifier.

Variable and variable declaration

Variables discussed here are different from the variables used in arithmetic expressions. They can be easily understood by considering them as a kind of box in which digits or characters are placed as shwon below. The types of variables are as shown below. Only the defined types of digits or characters can be assigned to each type of variables.

Each variable must be given an identifier called a variable identifier. Declarations of variable identifiers and the types of values to be assigned to them are made at the beginning of each PASCAL program.



Identifier

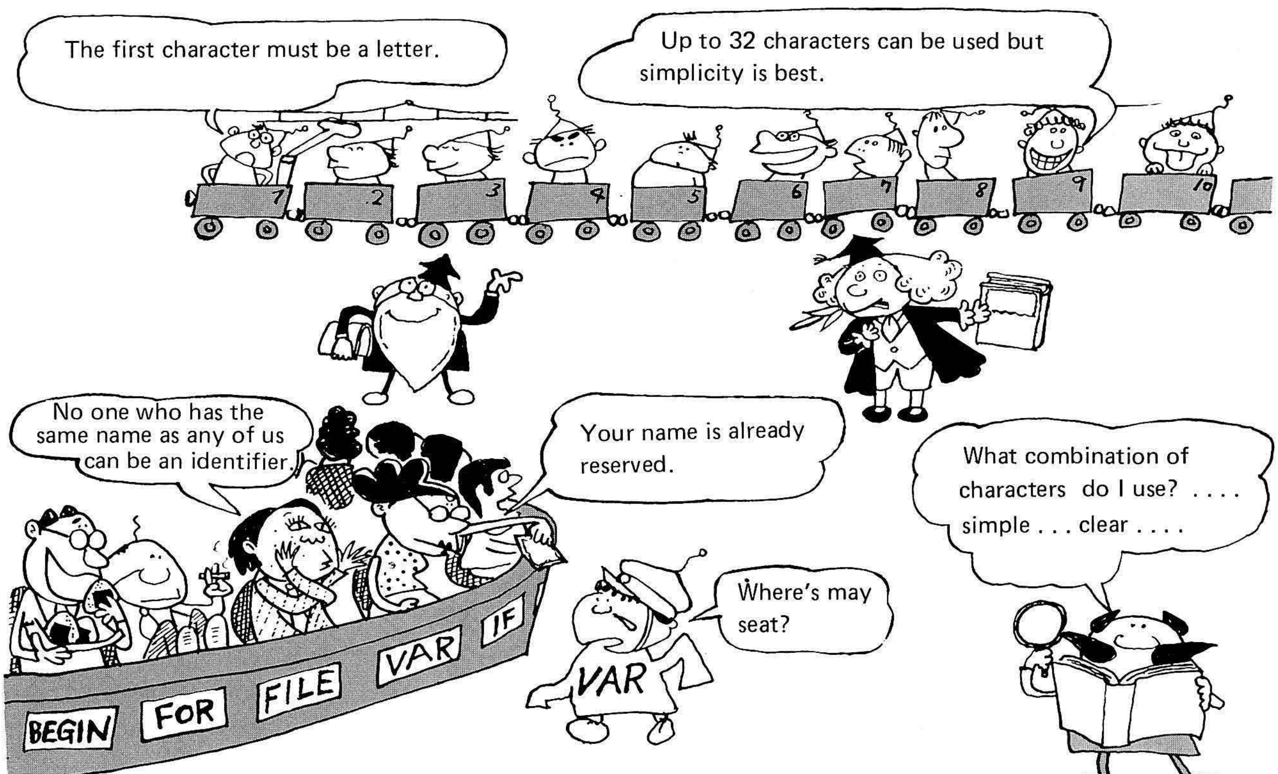
Various identifiers are used in PASCAL programs, and they must conform to the following rules.

(1) The first character of each identifier must be a letter (A through Z).
(2) The second and subsequent characters may be any combination of letters and digits.
(3) Reserved words cannot be used as identifiers.
(4) The maximum number of characters which can be used in each identifier is 32.

Reserved words are special words which are used for PASCAL instructions (such as **BEGIN**, **FOR**, **VAR**, READ, WRITE, etc.).

PASCAL interpreter SB-4515 recongnizes only lower case character for the PASCAL reserved words, statements, standard procedures, and standard functions. Although you may key them in upper case, the interpreter will display them in lower case.

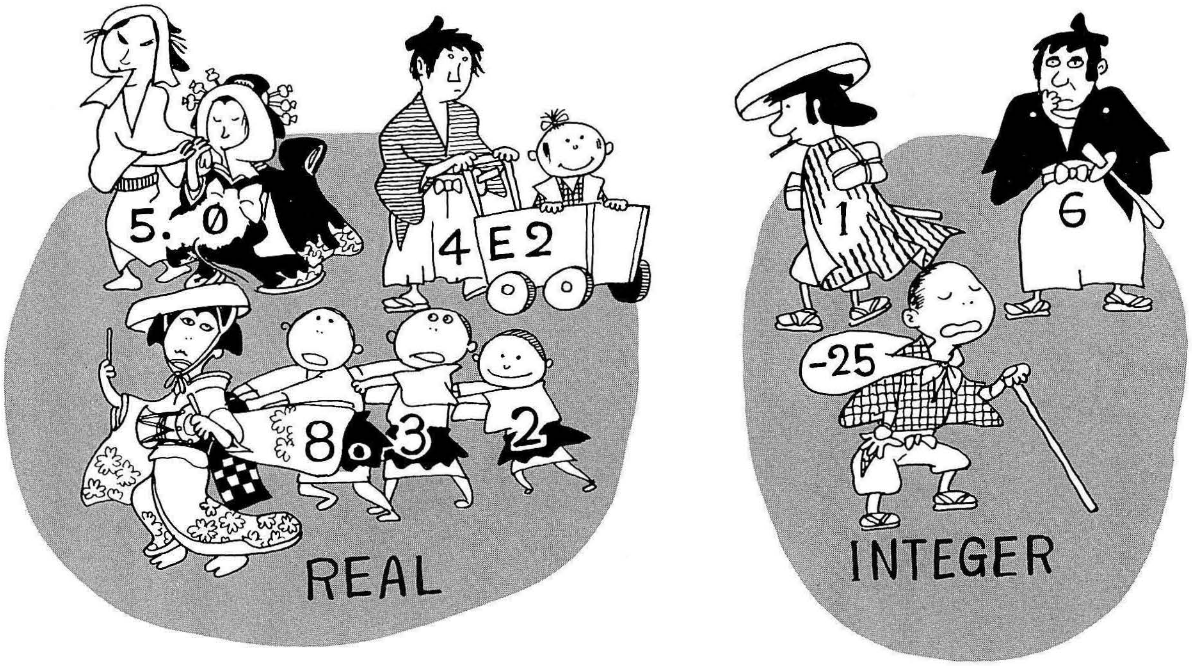
Only upper case alphanumeric characters are allowed, however, for variable names, array names, user procedure names, and user function names.



Integers and real numbers

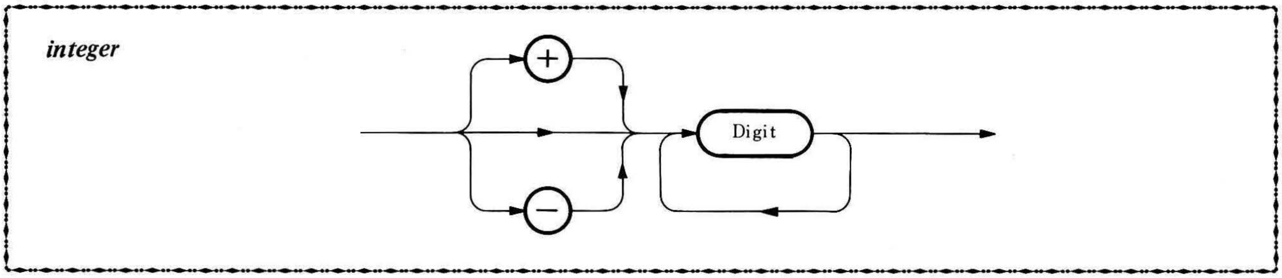
Mathematically speaking, integers are included in the group of real numbers. In PASCAL programs, however, they are treated separately. No real number can be assigned to an integer variable, and vice versa.

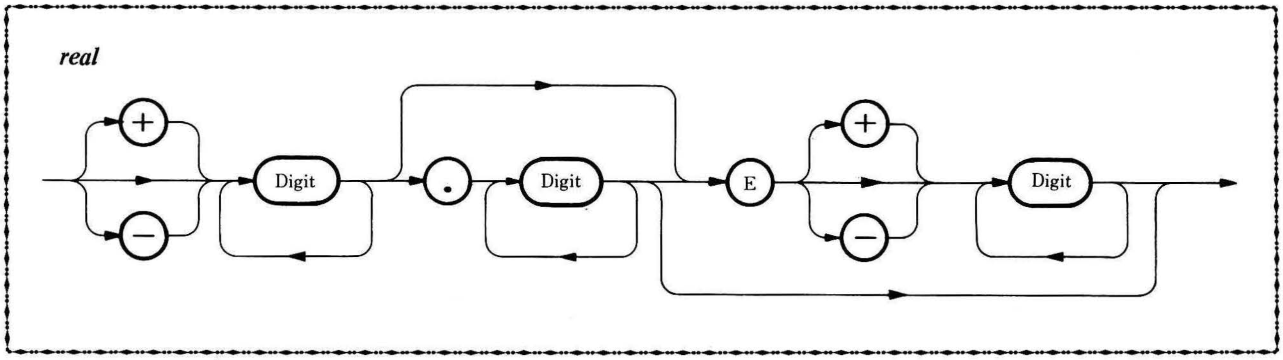
integer	0	1	−5	−25	3000		
real	0.0	1.0	−5.0	−25.0	3E3	8E−8	8.3E3



Syntax Diagrams for Integers and Real Numbers

The following are syntax diagrams for integers and real numbers. With the integer syntax diagram, +5, 3 and −25 are accepted, but 5., .5 and −3.2 are not. With the real syntax diagram, 5E10, +3.2 and −4.6 are accepted, but +5, .3 and 6 are not.



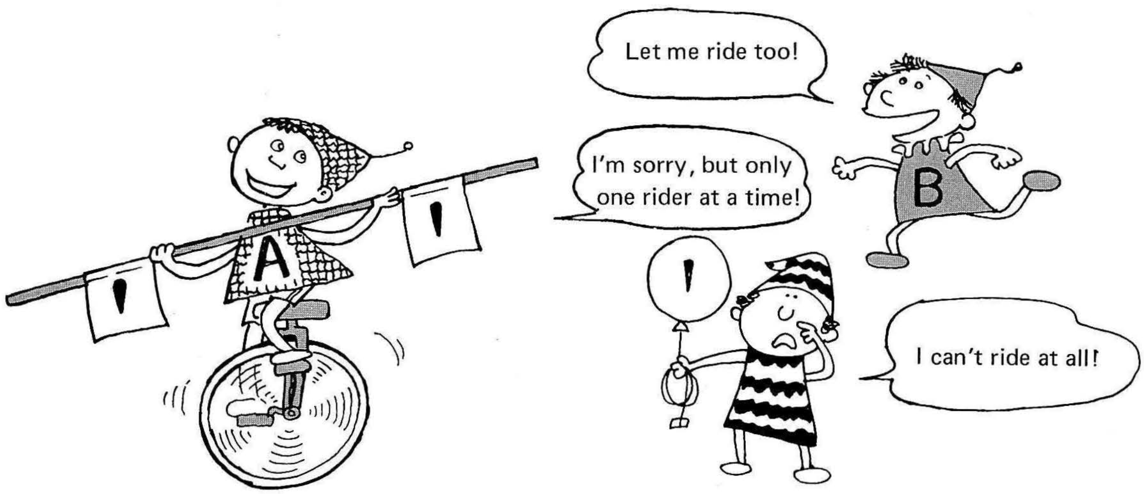


Character constants and character strings

Character Constants (*char* type data)

It may be necessary to assign a character value to a character variable or to compare one character with another one. Such a character value is called a character constant. Any of the characters shown in the ASCII code table on page 134 except the single quotation mark (') can be used as character constants. Such character data consists of a single character enclosed in single quotation marks.

Ex) 'A', 'B', '∗', '   ' . . . . Allowed
    ''', 'AB', 'A1' . . . . Not allowed



Character String

A character string is a set of characters enclosed in double quotation marks '"'. All characters shown in the ASCII code table on page 134 except for the double quotation mark can be used in character strings, including a single character.

Ex) "SHARP"   " C ⇨ ⇩ ∗∗ PASCAL ∗∗ "   "'"
    "TEL 06 621-1221"   "X" . . . . Allowed
    "INPUT" YES OR NO" " . . . . Not allowed (Double quotation marks used).

Separators

Separators are placed between variable identifiers, numbers and instructions to allow the computer to determine where each ends and begins.

Separators used in PASCAL programs are as follows:

(1) Space
(2) Comma (,)
(3) Semicolon (;)

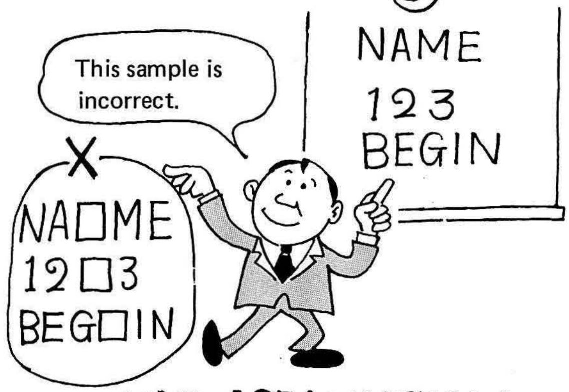
At least one separator must be placed between any two instructions, identifiers and numbers. A semicolon (;) is used to indicate the end of an instruction statement; an instruction statement may be written on more than one line, and only a semicolon (;) can be used to indicate the end of one. An identifier or expression, however, cannot be written on more than one line.

Ex)
```
var ↵
AREA ↵        } Allowed
: integer ↵
; ↵

va
R   AR ↵      } Not allowed
EA :  inte ↵
ger  ; ↵
```

IF□A=5□THEN

Spaces



VAR□ AREA:REAL;

At least one space is required. Two or more spaces may be used.

VAR□□□AREA□:□REAL;

These spaces may be permissable.

VAR AREA:INTEGER

No space

No semicolon (;)

VAR□ABC:INTEGER; ⟶ Declares variable ABC.

VAR□A,B,C:INTEGER; ⟶ Declares three variables: A, B and C.

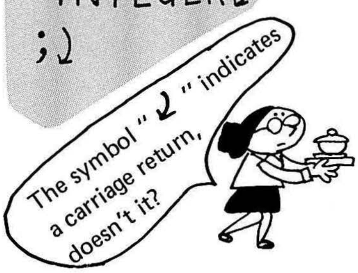
VAR□A□B□C:INTEGER; ⟶ A space cannot be used to separate two variables.

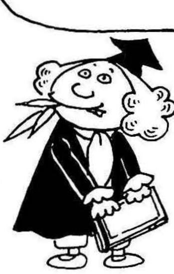
VAR ↵
AREA ↵
:INTEGER ↵
; ↵

In PASCAL programs a carriage return CR does not indicate the end of a statement.

In other words, the statement on the left is the same as that on the right.

VAR AREA : INTEGER;





Variable declaration

All variables used in a PASCAL program must be declared with the **var** declaration.
The **var** declaration begins with **var**, followed by a space. Variable identifiers and types follow these.

var □ A: *integer* ;

- variable type
- variable identifier
- variable declaration

A: precedes variable type declaration.
□ indicates a space.

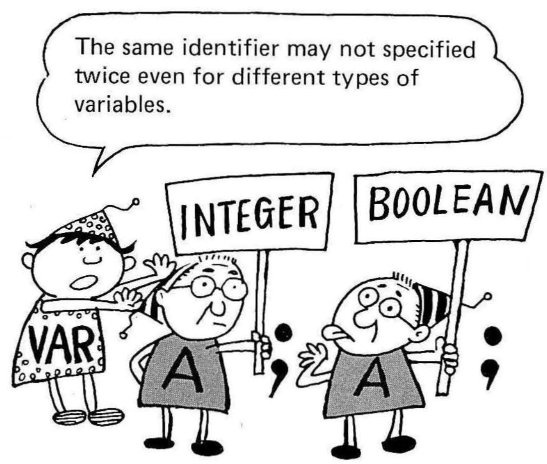
var	A: *integer* ;	Declares A as an *integer* type variable.
var	A, B, C: *real* ;	Declares A, B and C as *real* type variables.
var	A, B: *integer* ; X, DATA: *real* ;	Declares A and B as *integer* type variables and X and DATA as *real* type variables.
var	CH, SYMBOL: *char* ;	Declares CH and SYMBOL as *char* type variables.
var	Y, Z: *boolean* ;	Declares Y and Z as *boolean* type variables.

Note: *char* is an abbreviation for character.

File Declaration

It is possible to write data from variables onto cassette tape, or to enter data from cassette tape into variables, by specifying in the variable declaration that the variable is the counter part of a cassette tape file. This specification is called a file declaration.

Pages 35 and 36 explain the transfer of data between variable and cassette tape files, as well as the *read* and *write* statements.



(1) **var** X, Y : **file of** *integer* ;

- Type
- File declaration
- Identifiers

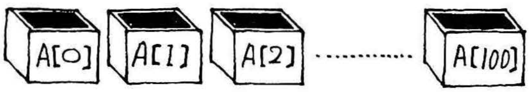
(2) **var** DATA : **file of** *real* ; A, B : *char* ;

var declaration (1) specifies tha variables X and Y are of the *integer* type and that they are counter parts of cassette tape files. **var** declaration (2) specifies that the variable DATA is of the *real* type and that it is the counter part of a cassette tape file; it also specifies that variables A and B are of the *char* type and have no relation with cassette tape files.

Array declaration

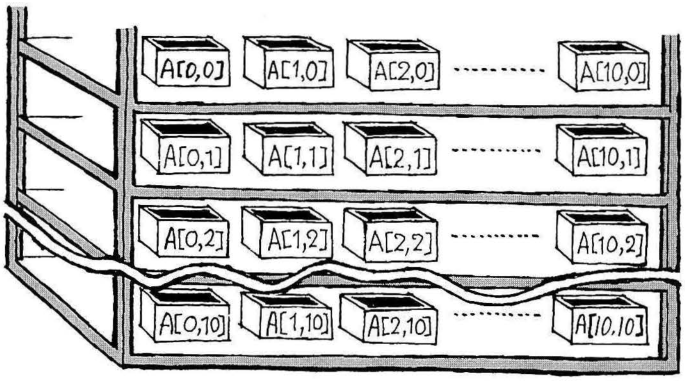
An array of variables is declared with an **array** declaration in the **var** declaration; this corresponds to the DIM statement in BASIC. There is no limit on the number of dimensions of array variables in PASCAL, except for the memory capacity. BASIC only provides for one and two-dimensional arrays.

One-dimensional array



A [100] One-dimensional array with 101 elements

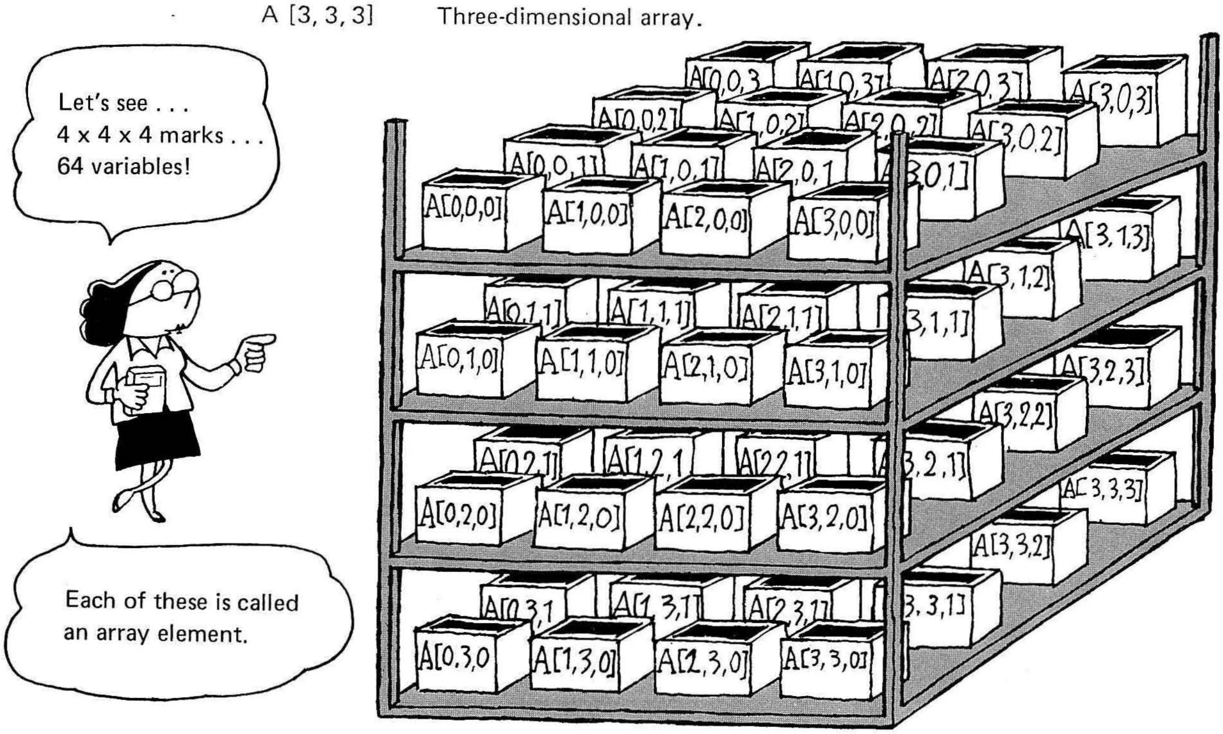
Two-dimensional array



A [10, 10] 11 x 11 two-dimensional array

Three-dimensional array

A [3, 3, 3] Three-dimensional array.



Array Declaration

An array variable is specified in the **var** declaration as follows.

var < array identifier > : **array** [index] **of** < element type > ;

Ex) **var** A : **array** [5] **of** *integer* ;

Specifies A as a one-dimensional array of *integer* variables with elements 0 through 5.

var TABLE : **array** [10, 10] **of** *char* ;

Specifies TABLE as a two-dimensional array of 11 x 11 *char* variables.

var DATA : **array** [10, 5, 5] **of** *real* ;

Specifies DATA are a three-dimensional array of *real* variables with 11 x 6 x 6 elements.

As shown in the above examples, the number of dimensions is determined by the number of indexes. An n-dimensional by specifying n indexes separated with commas. The size of arrays which can be specified differs according to the data type.

A sample program is shown below. In this program, the first two lines declare arrays and the third line declares variables.

var A : **array** [5] **of** *integer* ; TABLE : **array** [10, 10] **of** *char* ;
DATA : **array** [10, 5, 5] **of** *real* ;
X, Y : *real* ; Z : *boolean* ;

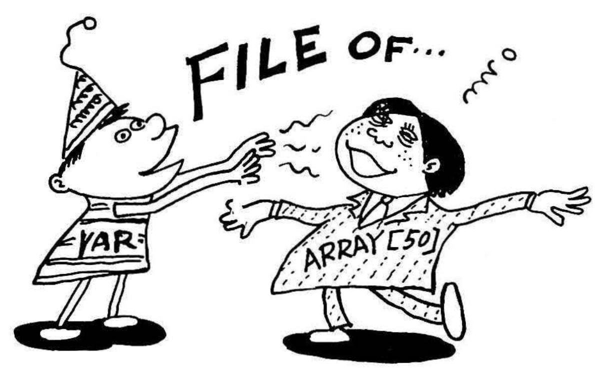
When the size and type of more than one array are the same, they are declared as follows.

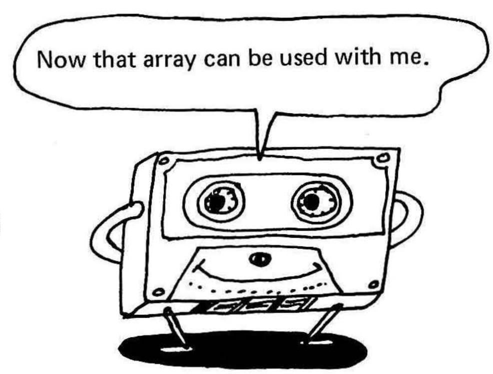
var X, Y, Z, DATA : **array** [15] **of** *real* ,

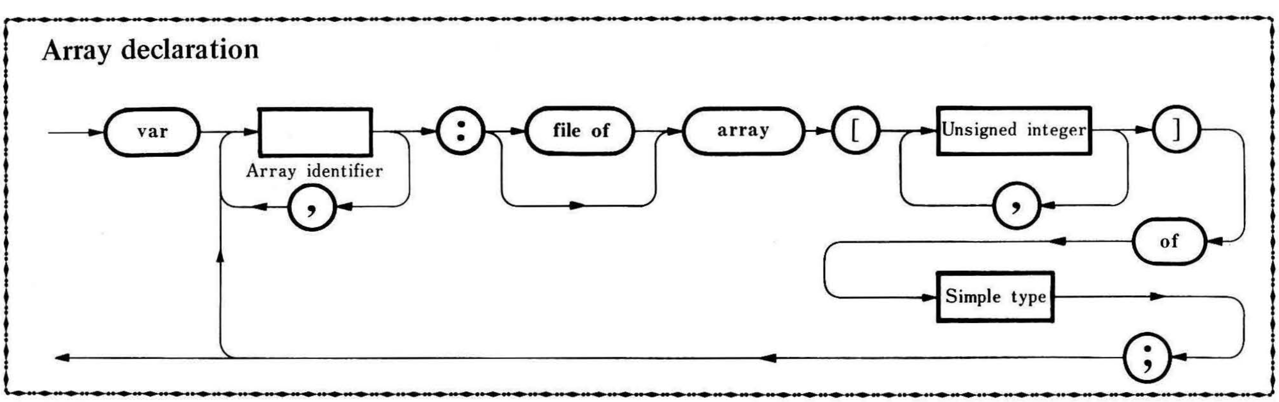
Files may be declared for arrays just as they may be for individual variables.

var DATA : **file of** **array** [50] **of** *real* ;

- *real* — Element type
- [50] — Index
- **file of** — File declaration
- DATA — Array identifier







Write and read array data to/from cassette tape

Let us code and execute a program which processes an array. The programming example is divided into two parts. The first part writes data in the array, then saves it in the cassette tape file; the second part reads the data from the cassette tape file back into an array and substitutes data from the array into variable X for display for each array element. First input the following. Instructions for the second part will be given later.

```
0. var DATA : file of array [25] of char;
1.      N, X : integer;
2. begin
3.    X : =65 ;
4.    for N : =0 to 25 do
5.      begin
6.        DATA [N] : =chr (X) ; ................ Assigns data to the array.
7.        X : =X+1
8.      end ;
9.    fname ( " ALPHABET " ) ;
10.   write (DATA[ ]) ; .............................. Saves the contents of the array in the cassette tape file.
11.   close
12 end
```

Run the program. If any errors exist, an error message will be displayed to request corrections.

(1) The *fname* statement opens the cassette file "ALPHABET" to allow array data to be written on the cassette tape.

(2) The *write* statement at line 10 automatically saves the contents of the array DATA [ ] in the cassette tape file.

(3) The cassette tape stops when recording is completed. The *close* statement closes the cassette file. The system displays "Ready." on the CRT screen when the program terminates after recording is completed.

Rewind the cassette tape and clear the program executed. Then, input the following program.

```
0. var DATA : file of array [25] of char;
1.      N : integer ; X : char ;
2. begin
3.    fname ( " ALPHABET " ) ;
4.    read (DATA[ ]) ; ................................Reads data from the cassette tape into the array.
5.    close ;
6.    for N : =0 to 25 do
7.      begin
8.        X : =DATA[N] ; ............................ Assigns data to X.
9.        write (X : 4) ................................ Displays data in X.
10.     end
11. end.
```

When the above program is executed.

(1) The *fname* statement opens the cassette file "ALPHABET", enabling the system to read data from the cassette tape.

(2) The *read* statement at line 4 automatically reads data, and assigns it in succession to the array elements.

(3) After data has been read, the tape stops.

(4) The letters A through Z are displayed on the CRT screen.

As shown in the example on the preceding page, a data file can be created by using the **file** declaration. A distinct name to indicate its contents must be given to each data file.

Pay attention to the following when inputting or outputting array data to or from the cassette tape file. To write array data in the file, use

write (< array identifier > [ ])

and to read array data from the file, use

read (< array identifier > [ ])

The symbols " [" and "] " must be entered without any intervening spaces or characters. As indicated above, it is impossible to write or read just one element into an array from the cassette file with specifications such as *write* (DATA [5]) or *read* (DATA [5]).

File arrays are written on or read from the cassette file in blocks of the size specified for each array.

The following does not illustrate a normal situation, but it may be used.

Consider a one-dimensional array which has undergone the **file** declaration, A [59] . This array can be written on the cassette tape with *write* (A [ ]).

Now consider a two-dimensional array, B [19, 2] , which has 20 x 3 elements. Both arrays have the same number of elements, so array data written in the cassette file from array A can be read into array B with *read* (B [ ]). The data type of both arrays must be the same.

Data can be transferred between arrays in this manner if the number of elements of both arrays is the same and both array variable types are the same.

Array X [5] is assigned with *char* data as follows, then written in the cassette tape file.

X [0] = 'A' X [1] = 'B' X [2] = 'C'
X [3] = 'D' X [4] = 'E' X [5] = 'F'

When this data is read into two-dimensional array [2, 1] , it is assigned as follows.

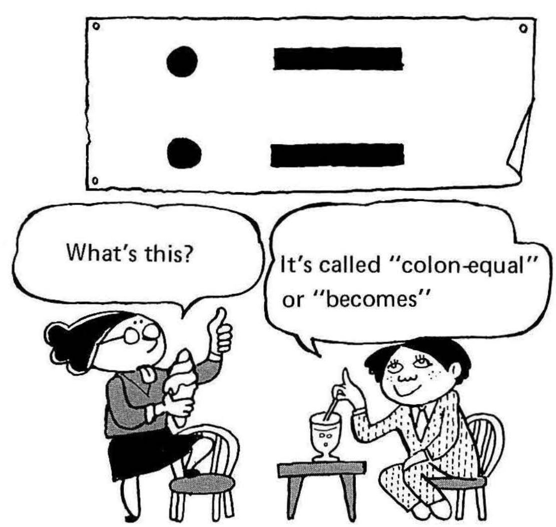
Y [0, 0] = 'A' Y [1, 0] = 'B' Y [2, 0] = 'C'
Y [0, 1] = 'D' Y [1, 1] = 'E' Y [2, 1] = 'F'

Note: Instruction words which have not been explained in this and subsequent examples will be explained in detail later.
The following are instruction words which appear frequency.

read, *readln* . . . . Data input instructions which are similar to the INPUT statement in BASIC.

write, *writeln* . . . . Data output instructions which are similar to the PRINT statement in BASIC.

A: = X + Y . . . . : = indicates assigns the sum of X and Y to A.



Chapter 4

Data and Expressions

The basic types of data used in PASCAL programs are

> ***integer,***
> ***real,***
> ***boolean,*** *and*
> ***char.***

Any combination of data and operators is called an expression.

Integer expressions

The following are the five integer operators. All integer expressions are formed of integer operators and *integer* data.

Precedence	Operator	Operation	Format	Example	Result
1	*	Multiplication	A*B	5*2	10
1	**div**	Division with truncation	A **div** B	5 **div** 2	2
1	**mod**	Modulus	A **mod** B	5 **mod** 2	1
2	+	Sum	A+B	5+2	7
2	–	Subtraction	A–B	5–2	3

div gives a truncated integer result. For example,

X : = 10 **div** 3 . . . . 10 ÷ 3 = 3 with the remainder 1. 3 is assigned to X.

X : = 15 **div** 7 . . . . 15 ÷ 7 = 2 with the remainder 1. 2 is assigned to X.

mod gives the remainder. For example,

X : = 10 **mod** 3 . . . . 10 ÷ 3 = 3 with the remainder 1. 1 is assigned to X.

X : = 17 **mod** 7 . . . . 17 ÷ 7 = 2 with the remainder 3. 3 is assigned to X.

Note the following when writing an integer expression.

(1) A+B is a correct expression, and the following expressions are also correct:
A + B, A+ B, A +B and A + B.

(2) A–B is a correct expression, and the following expressions are also correct:
A – B, A– B, A –B and A – B.

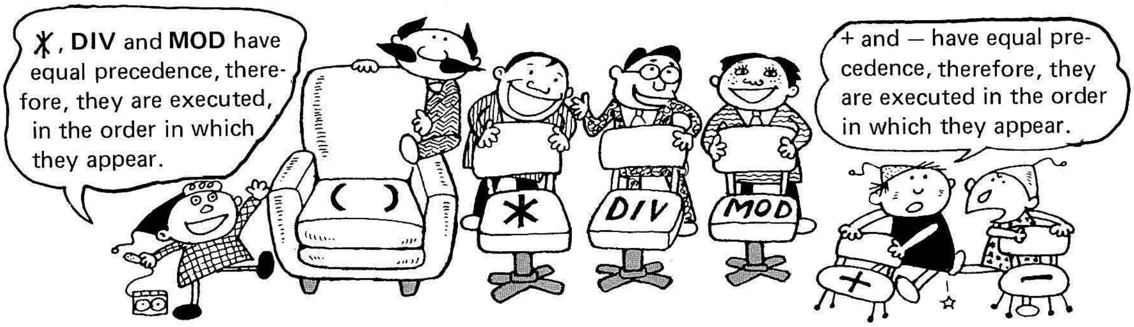
(3) A **div** B is a correct expression, but **AdivB**, A **divB** and **Adiv** B are incorrect.
A **div** B is correct.

(4) A **mod** B is a correct expression, but **AmodB**, A **modB** and **Amod** B are incorrect.
A **mod** B is correct.

Be sure to insert a space before and after **div** (or **mod**).

Precedence of Operators

The precedence of operators in an arithmetic expression is shown in the figure below.



The following are examples of integer operations; familiarize yourself with how these are performed.

3+5 **div** 2 gives 5.
(3+5) **div** 2 gives 4.

60–6*8+2 gives 14.
(60–6)*8+2 gives 434.

6+(6*(3–1)) gives 18.
(6+6)*3–1 gives 35.

9–7 **mod** 2 gives 8.
9*7 **mod** 5 gives 3.

80 **mod** 9 **div** 5 gives 1.
80 **div** 9 **mod** 5 gives 3.

3+6*(9 **div** 2) **mod** 2 gives 3.
(3+6)*9 **div** 2 **mod** 3 gives 1.

A – sign appearing in an integer expression is always executed as a – operator. Thus, –28 **div** –3 is incorrect because two integer operators, **div** and –, appear consecutively. –28 **div** (–3) is a correct expression and gives the same result as –(28 **div** (–3)).

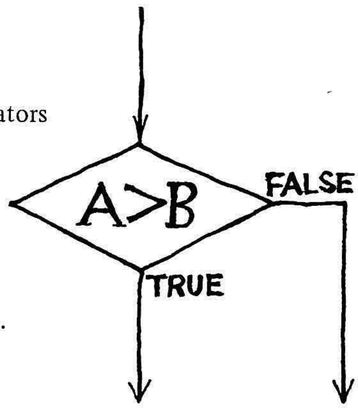
–28 **mod** –3 is also an incorrect expression. –28 **mod** (–3) is correct, and gives the same result as –(28 **mod** (–3)).

Familiarize yourself with the following:

(–15) **div** 8	gives –1.	(–15) **mod** 8	gives –7.
(–28) **div** (–3)	gives 9.	(–28) **mod** (–3)	gives –1.
56 **div** (–9)	gives –6.	56 **mod** (–9)	gives 2.
–10 **div** 15	gives 0.	–10 **mod** 15	gives –10.

Relational Operators

Relational operators are used for comparing two data values. The relational operators used in integer expressions are shown below.



=	checks whether the left member is equal to the right member.
< >	checks whether the left member is inequal to the right member.
< =	checks whether the left member is equal to or less than the right member.
> =	checks whether the left member is equal to or greater than the right member.
<	checks whether the left member is less than the right member.
>	checks whether the left member is greater than the right member.

The result is always *true* or *false*. For example, A > B gives *true* when A is greater than B. This is shown by the flow chart at right.

Only one relational operator can be used in an expression; X < > Y = Z is an incorrect expression because it contains two relational operators.

Boolean expressions

Boolean expressions are used for making decisions, **YES** or **NO**. The only two values which may be given by a Boolean expression are *true* and *false*. Four Boolean operators are provided for use in Boolean expressions.

These are also called logical operators.

Precedence	Operator	Meaning	Example
1	**not**	Logical NOT	**not** (A=B) gives *true* when A is not equal to B.
2	**and**	Logical AND	(A>B)**and**(A>C) gives *true* when A is greater than both B and C.
3	**or**	Logical OR	(A>B)**or**(A>C) gives *true* when A is greater than B or C.
3	**xor**	Exclusive OR	(A>B)**xor**(A>C) gives *false* when A is greater than both B and C, or when A is less than both B and C, and gives *true* when A is greater than B and less than C, or when A is less than B and greater than C.

not A is *true* if A is *false*; otherwise it is *false*.
A **and** B is *true* if both A and B are *true*; otherwise it is *false*.
A **or** B is *true* if either or both A and B are *true*; otherwise it is *false*.
A **xor** B is *true* if A and B have different Boolean values; otherwise it is *false*.

These operations may not be familiar, but they are necessary when using computers.

An exercise follows.

Obtain the results of **not** A, A **and** B, A **or** B and A **xor** B where

(1) both A and B are *true*.
(2) A is *true* and B is *false*.
(3) A is *false* and B is *true*, and
(4) both A and B are *false*.

The answers are given on the next page.

Expressions such as **notA**, **AandB**, **AorB** and **AxorB** are incorrect.

Precedence

The precedence of Boolean operators is as follows.

Highest	**not**	
	and	
Lowest	**or**	**xor**

Relational operators can be used in conjunction with Boolean operator in an expressions. The precedence of relational operators is lower than that of Boolean operators. Two or more Boolean operators may be used in an expression. For example, A **xor** B **and** C is a correct expression. In this case, **and** is applied before **xor** because of the precedence, that is, first B **and** C is executed, then its result and A are subjected to the exclusive OR operation. Therefore, when it is necessary to first apply **xor** to A and B, the expression must be written as (A **xor** B) **and** C.

Since **or** and **xor** have equal preceedence, the one which appears earliest is applied first.

In the case of A **and not** B, **not** B is executed first because it has higher precedence, then its result and A are subjected to **and**.

Great care must be taken when combining relational operators and Boolean operators, or an unexpected result may be obtained.

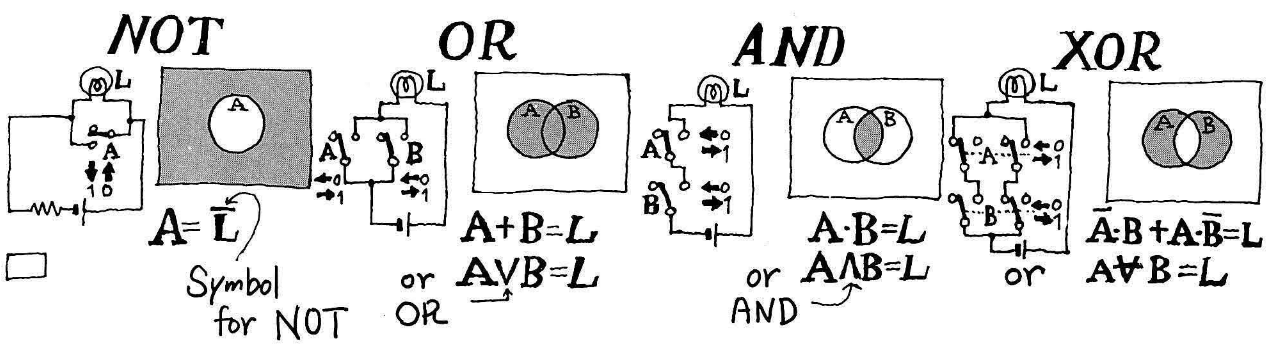
A > 0 **and** A < 100 is not correct.

Use parentheses as follows.

(A > 0) **and** (A < 100)

Solutions for exercise

	(1)	(2)	(3)	(4)
not A	*false*	*false*	*true*	*true*
A **and** B	*true*	*false*	*false*	*false*
A **or** B	*true*	*true*	*true*	*false*
A **xor** B	*false*	*true*	*true*	*false*



Real expressions

The four operators shown below are used in real expressions. Constants and variables used in real expressions all must be *real.*

Precedence	Operator	Meaning	Example
1	*	Multiplication	A*B
1	/	Division	A/B
2	+	Addition	A+B
2	–	Subtraction	A–B

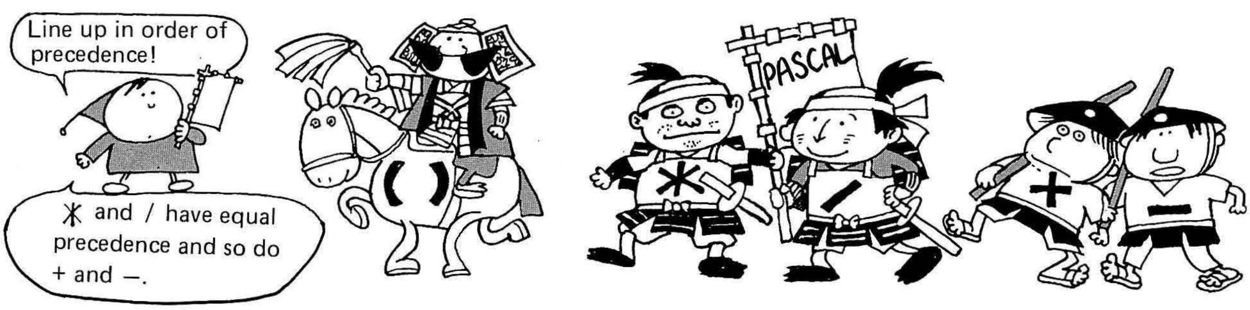
The operators **div** and **mod** used in integer expressions are not used in real expressions. ^ power used in BASIC is not provided in PASCAL.

All constants or variables processed by real expressions must be *real*; therefore the result cannot be assigned to any *integer* variable even if it has the form of an integer (e.g. 2 or 3) since it is *real* (2.0 or 3.0).

When **var** A: *integer*; B: *real*; is declared in the **var** declaration, the following expressions cannot be excuted.

A+B A*B

In practice, however, it may be necessary to assign an *integer* value to a *real* variable, or vice versa. Instruction which convert one type of value into the other are porvided for this purpose. These instructions will be explained in the section on "Standard Functions."



All relational operators, =, <>, <=, >=, <, > may be used in real expressions; their meanings are the same as they are in integer expressions. Both members of the expression must be *real.*

When variable A and B are *real*, neither A **and** B nor A **or** B can be executed. However, the following can be executed because expressions using relational operators give Boolean results.

	not	(A>B)	A space is not always required between a Boolean operator and the parentheses surrounding relational expressions.
(A>B)	**and**	(A>C)	
(A>B)	**or**	(A>C)	
(A>B)	**xor**	(A>C)	

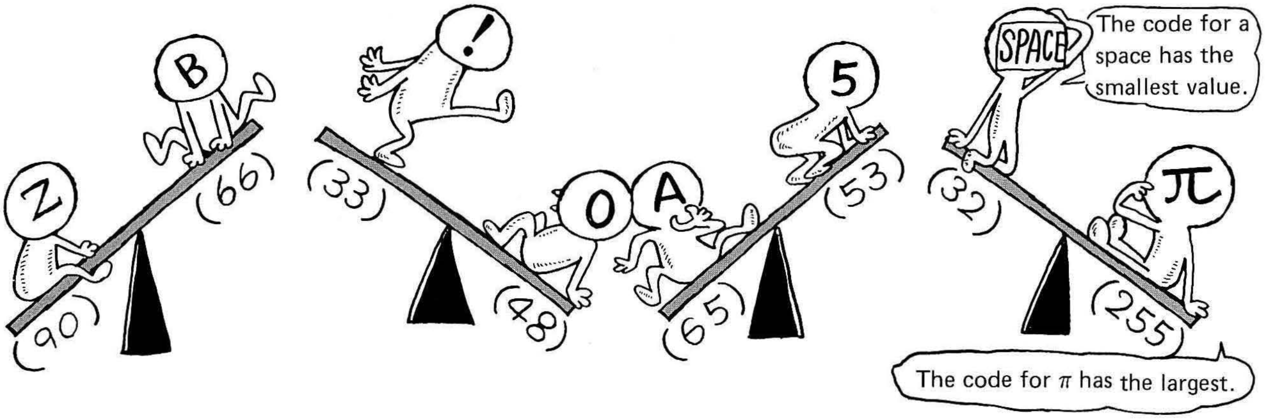
CHAR expressions

char variables are similar to the string variables of BASIC, but only logical operations using relational operators can be applied to them.

If CHA and CHB are declared as *char* variables and 'A' is assigned to CHA and 'B' is assigned to CHB, when

CHA=CHB

is executed, the character code for A is compared with that for B. Since their codes are 65 and 66, respectively (see the code table on page 134), *false* results. Any of the relational operators, =, <>, <=, >=, <, >, may be used in such expressions.



Logical operators cannot be applied directly to *char* variables.
When CHA, CHB and CHC are *char* variables,

CHA **and** CHB . . . . cannot be executed

(CHA > CHB) **and** (CHA > CHC)
not (CHB <= CHC)
(CHA = CHB) **xor** (CHAR > CHC)

} . . . . can be executed because the expressions in parentheses give Boolean results.

Special instructions relating to *char* data will be explained later.

Standard functions

A function performs a prescribed task and returns a result when data is applied to it. A function which performs a task which is predefined is called a standard function. Several standard functions are provided in PASCAL.

When a variable used in a function is enclosed in parentheses, it is called a formal parameter; the value assigned to a formal parameter is called an actual parameter. No file identifier can be used as a parameter in any standard function. The standard functions in PASCAL are described below.

1. ODD (X)

The parameter specified in this function must be an *integer* value and a *boolean* result is obtained.
This function gives *true* if the parameter is odd, otherwise it gives *false*.

A : = *odd* (5) *true* is assigned to variable A.
A : = *odd* (6) *false* is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

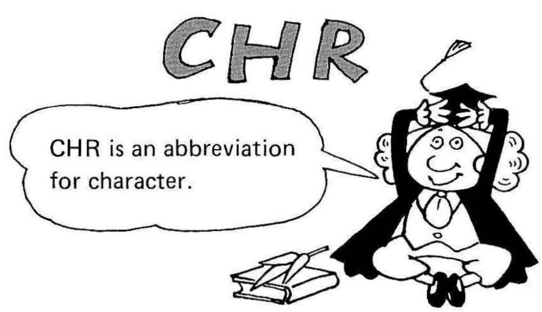
2. CHR (X)

The parameter specified in this function must be an *integer* value and a *char* value is obtained as the result.

This function gives the character whose code value is specified in the parameter. It corresponds to CHR$ (X) in BASIC.

A : = *chr* (8Ø) The character 'P' is assigned to variable A.

Any constant, variable or expression may be used as the parameter.



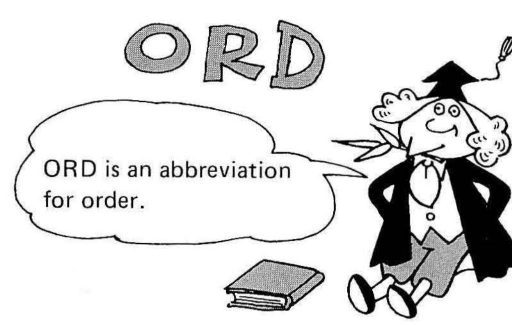
3. ORD (X)

The palameter specified in this function must be a *char* value and an *integer* value is obtained as the result.

This function gives the *integer* value corresponding to the code for the character specified in the parameter.

A : = *ord* ('X') 88 (the code for 'X') is assigned to variable A.

Any constant, variable or expression may be used as the parameter.



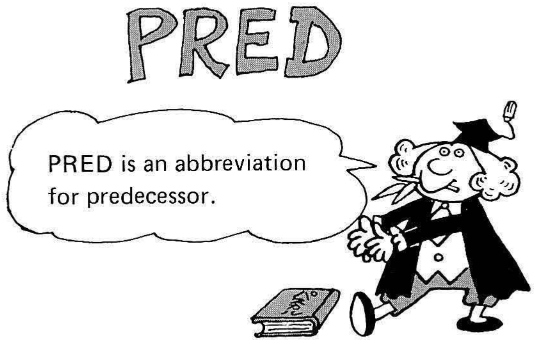
4. PRED (X)

The parameter specified in this function must be a *char* value and a *char* value is obtained as the result.

This function gives the character which has the same code value as that of the character specified in its parameter minus 1.

A : = *pred* ('Y')   The character 'X' is assigned to variable A.

Any constant, variable or expression may be used as the parameter.



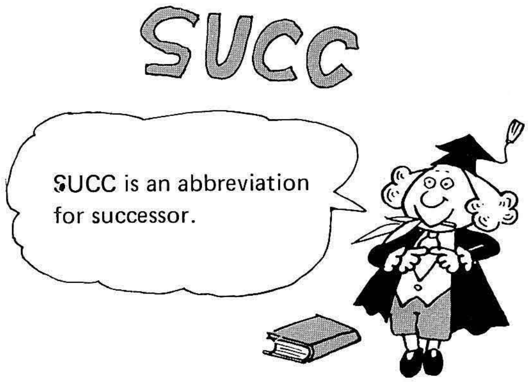
5. SUCC (X)

The parameter specified in this function must be a *char* vaue and a *char* value is obtained as the result.

This function gives the character which has the same code vaue as that of the character specified in its parameter plus 1.

A : = *succ* ('Y')   The character 'Z' is assigned to variable A.

Any constant, variable or expression may be used as the parameter.



Inverse Functions

Of these functions, *chr* is the reverse of *ord* and *pred* is the reverse of *succ.* It is said that one is the inverse function of the other.

The relationship between inverse functions can be understood from the following examples.

chr (*ord* ('X')) = X	*ord* ('X') gives 88 and *chr* (88) gives 'X'.
pred (*succ* ('Y')) = Y	*succ* ('Y') gives 'Z' and *pred* ('Z') gives 'Y'.
ord (*chr* (88)) = 88	*chr* (88) gives 'X' and *ord* ('X') gives 88.
succ (*pred* ('Z')) = Z	*pred* ('Z') gives 'Y' and *succ* ('Y') gives 'Z'.

6. TRUNC (X)

The parameter specified in this function must be a *real* value and an *integer* value is obtained as the result.

This function converts *real* data values into *integer* data values.

A : = *trunc* (3.14)   The *integer* value 3 is assigned to variable A.

A : = *trunc* (–2.8)   The *integer* value –2 is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

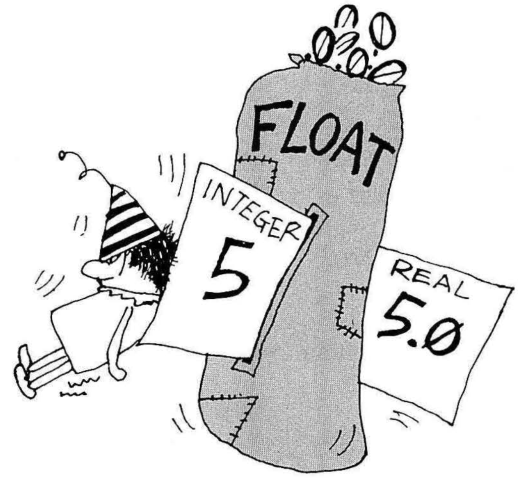
7. FLOAT (X)

The parameter specified in this function must be an *integer* value and a *real* value is obtained as the result.

The function is the inverse of the *trunc* function; it converts *integer* data values to *real* data values.

A : = *float* (15)	*real* value 15.0 is assigned to variable A.
A : = *float* (–8)	*real* value –8.0 is assigned to variable A.
B : = *float* (*trunc* (3.14))	*real* value 3.0 is assigned to variable B.

Any constant, variable or expression may be used as the parameter.



8. ABS (X)

The result is a *real* value when value specified in the parameter is *real*; the result is an *integer* value when the value specified in the parameter is an *integer* value.

This function gives the absolute value of the value specified in the parameter, just like the ABS (X) function in BASIC.

A : = *abs* (–3.5)	*real* number 3.5 is assigned to variable A.
B : = *abs* (–365)	*integer* number 365 is assigned to variable B.

Any constant, variable or expression may be used as the parameter.

9. SQRT (X)

The parameter specified in this function must be a *real* value which is greater than or equal to zero. The result is a *real* value.

This function gives the square root of the value specified in the parameter. Any constant, variable or expression may be used as the parameter.

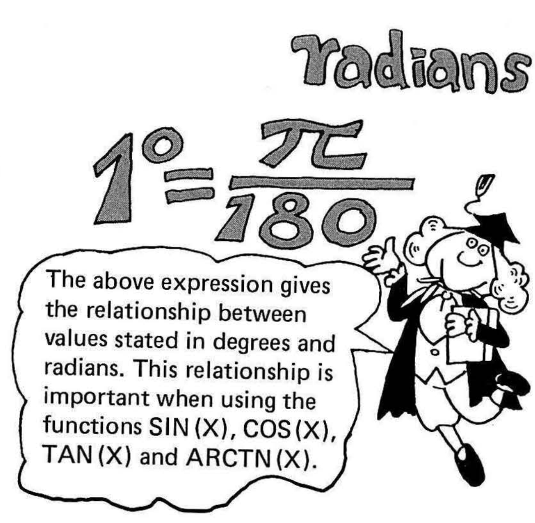
10. SIN (X)

The parameter specified in this function must be a *real* value (expressed in radians) and a *real* value is obtained as the result. This function gives the sine of the value specified in the parameter.

To obtain the sine of a value stated in degrees, first convert the value to radians. For exmaple, to obtain sin 30°, specify

A : = *sin* (30.0∗ 3.1415927/180.0)

Any constant, variable or expression may be used as the parameter.



11. COS (X)

The parameter specified in this function must be a *real* value (in radians) and a *real* value is obtained as the result.

A : = *cos* (200.0 ∗ 3.1415927/180.0) The value of cos 200° is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

12. TAN (X)

The parameter specified in this function must be a *real* value (in radians) and a *real* value is obtained as the result.

A : = *tan* (30.0 ∗ 3.1415927/180.0) The value of tan 30° is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

13. ARCTAN (X)

The parameter specified in this function must be a *real* value and a *real* value between $-\pi/2 \sim \pi/2$ (in radians) is obtained as the result.

A : = *arctan* (X) The value of $\tan^{-1}$X in radians is assigned to variable A.
A : =180.0/3.1415927 ∗ *arctan* (X) The value of $\tan^{-1}$X in degrees is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

14. EXP (X)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result. This function gives the value of e^x, where e=2.7182818.

A : = *exp* (1.0) 2.7182818 is assigned to variable A.
A : = *exp* (0.0) 1.0 is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

15. LN (X)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result. This function gives the value of $\log_e X$, where $X>0$.

A : = *ln* (3.0) 1.0986123 is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

16. LOG (X)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result. This function gives the value of $\log_{10} X$, where $X>0$.

A : = *log* (3.0) 0.47712125 is assigned to variable A.

Any constant, variable or expression may be used as the parameter.

17. RND (X)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result.

This function generates pseudo-random numbers between 0.00000001 and 0.99999999, and works in two manners depending on the value specified as the parameter.

When the value specified as the parameter is larger than 0, the function gives the pseudo-random number next to the one previously given in the pseudo-random number group. When the value is 0 or negative, the function generates a pseudo-random number group and gives its initial value.

A : = *rnd* (1.0) A : = *rnd* (3.0)	A pseudo-random number which has no relation to the parameter value is assigned to variable A.
A : = *rnd* (0.0) B : = *rnd* (−3.0)	The same value is assigned to both variables A and B.

Any constant, variable or expression may be used as the parameter.

18. PEEK (X)

The parameter specified in this function must be an *integer* value and a *char* value is obtained as the result.

This function gives a code (0~255) which corresponds to data stored in the address specified (in decimal) by the parameter.

A : = *peek* (4608) The data code stored in address 4608 is assigned to variable A.

Any constant, variable or expression may be used as the parameter. Use the *ord* function to obtain the result as an *integer* value, as in B: = *ord* (*peek* (4608)).

This function is corresponding to PEEK(X) in BASIC.

19. CIN

This function has no parameter, and a *char* value is obtained as the result. This function gives the ASCII code which corresponds to the character in the position on the CRT screen at which the cursor is located.

A : = *cin* The ASCII code of the character displayed at the cursor position is assigned to variable A.

20. INPUT (X)

The parameter specified in this function must be an *integer* value and a *char* value is obtained as the result.

This function reads data on the port specified by the parameter. For port specification, refer to the explanation of the *output* statement on page 84.

This function executes machine language, $ED78, (i.e. IN A, (C)). The value of X is loaded in the BC register and data is read into the accumulator.

Any constant, variable or expression may be used as the parameter.

A : = *input* (255) Data on port 255 ($FF) is read into variable A. To obtain data of type *integer*, use A: = *ord* (*input* (255)).

21. KEY

This function has no parameter, and a *char* value is obtained as the result. This function gives the ASCII code corresponding to that of the key being pressed. If no key is pressed when this function is executed, the code corresponding to zero is obtained.

A : = *key* The ASCII code corresponding to the being pressed is assigned to variable A. When no key is depressed, the code corresponding to zero is assigned to A.

(*ord* (*key*) gives zero when no key is depressed.)

22. CSRH

This function has no parameter, and an *integer* value is obtained as the result. The *integer* value indicates the current location of the cursor on the horizontal axis. The cursor position changes each time the *cursor*, *write*, *writeln*, *read* or *readln* statement is executed, and its X-coordinate is given by this function.

The value of this function takes stays within the following range for each character display mode:

80-character mode: 0 ≦ *csrh* ≦ 79

40-character mode: 0 ≦ *csrh* ≦ 39

23. CSRV

This function has no parameter, and an *integer* value is obtained as the result in the same manner as the *csrh* function. The value indicates the current location of the cursor on the vertical axis and takes stays within the following range for both character modes mentioned above:

$0 \leqq csrv \leqq 24$

24. POSH

This function has no parameter, and an *integer* value is obtained as the result. The *integer* value indicates current location on the horizontal axis of the position pointer in the graphic display area. The position pointer moves each time the *position* or *pattern* statement is executed, and its X-coordinate is given by this function.

The value takes stays within the following range:

$0 \leqq posh \leqq 319$

25. POSV

This function has no parameter, and an *integer* value is obtained as the result in the same manner as the *posh* function. The value indicates the current location on the vertical axis of the position pointer in the graphic display area and takes stays within the following range:

$0 \leqq posv \leqq 199$

26. POINT (X, Y)

This function has two parameters which must be *integer* values, and an *integer* value is obtained as the result. The value is indicating whether the dot (X, Y) in the graphic display area is set or reset.

Result of the *point* function	Point information
0	Points in both graphic areas 1 and 2 are reset.
1	Only point in graphic area 1 is set.
2	Only point in graphic area 2 is set.
3	Points in both graphic areas 1 and 2 are set.

Chapter 5

Statements

A **Statement** *is an unit of execution of a PASCAL program. There are two types of statement.*

Simple Statement . . . .*Statement which cannot be grammatically divided*

Structured Statement . . . . . .*A statement which consists of multiple simple statements.*

Assignment statement

An **assignment statement** assigns a value to a variable, function identifier or array. This statement cannot be grammatically divided, so it is called a simple statement.

Variable.: =< expression > ;

Ex) X : =A+B ; The value previously assigned to A is added to the value previously assigned to B and the result is placed in X.

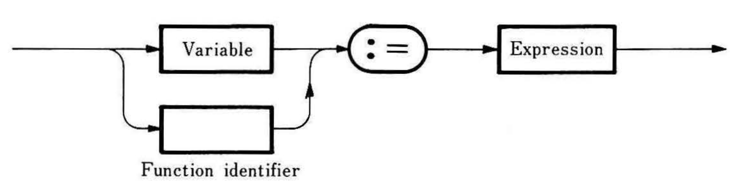
: = is called the assignment operator. The type of the left member must be the same as that of the right member.

A : =5 ;	Assigns 5 to variable A. A must be an *integer* variable.
B : =5.0 ;	Assign 5.0 to variable B. B must be a *real* variable.
C : =*true* ;	Assigns the logical value *true* to C. C must be a *boolean* variable.
D : ='A' ;	Assigns 'A' to D. D must be a *char* variable.
E : =(X > 0) **AND** (Y > 0)	E must be a *boolean* variable and X and Y must be *integer* values. *true* is assigned to E when both X and Y are positive, otherwise *false* is assigned to E.

Pay attention to the data type, especially when assigning a constant value to a variable. Review the data types of constants with the following examples.

integer constants	*real* constants
0	0.0
5	5.0
-15	-15.0
123	123.0
1000	1000.0 or 1E+3
–	0.35
–	0.01 or 1E+2

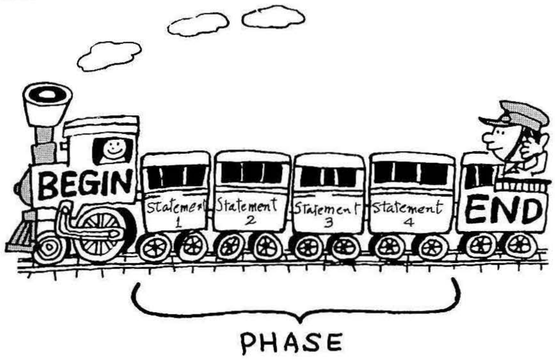
Assignment statement



Compound statements

A PASCAL program section consisting of several statements which are surrounded with **begin** and **end** is called a phase. A compound statement is formed of a phase, **begin** and **end**.

The executable section of a PASCAL program always consists of a combination of compound statements. The following sample program gives the Fahrenheit value of a temperature stated in degrees Centigrade using the equation, F = 1.8C+32.

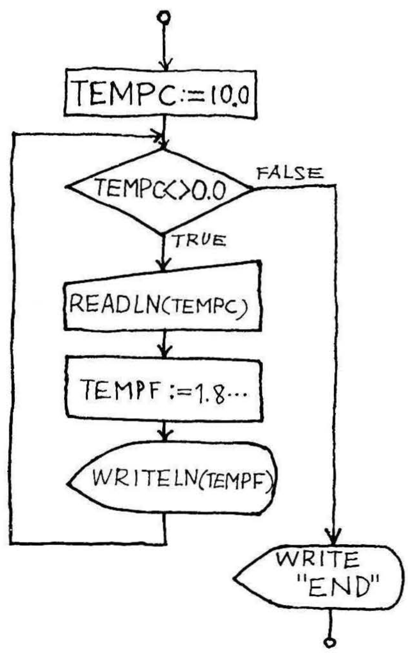



```
var TEMPF, TEMPC : real;
    begin
        readln (TEMPC) ;
        TEMPF : =1.8*TEMPC+32.0 ;
        writeln (TEMPF)
    end.
```

readln (TEMPC) ; ··· Reads a temperature stated in degree Centigrade.
TEMPF : =1.8*TEMPC+32.0 ; ··· Calculates the Fahrenheit equivalent
writeln (TEMPF) ··· and outputs the result.
Phase

A phase may include another phase as shown below. This is referred to as **block structure.** There is no limit on the number of levels of phases.

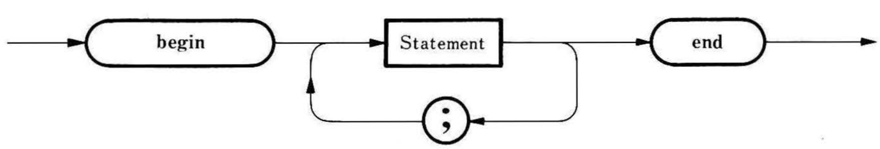
```
0. var TEMPC, TEMPF : real;
1.     begin
2.         TEMPC : =10.0 ;                              phase 1
3.         while TEMPC<>0.0 do
4.             begin
5.                 readln (TEMPC) ;                     phase 2
6.                 TEMPF : =1.8*TEMPC+32.0 ;
7.                 writeln (TEMPF)
8.             end ;
9.         write (" END ")
10.    end.
```



The above program operates as indicated in the flowchart at right.

(1) Line 2 is a dummy entry which starts the loop.
(2) Line 5 reads data from the keyboard.
(3) Line 6 calculates the Fahrenheit temperature.
(4) Line 7 outputs the result.
(5) When the data read is other than 0.0, lines 5 through 7 are executed again ; when it is 0.0, END is displayed and the program ends.

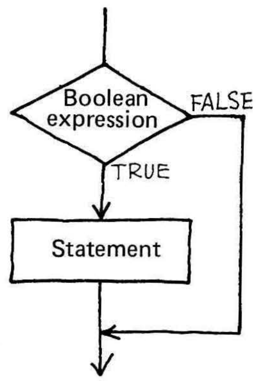
Compound statement



IF statement (choice)

An **if** statement chooses one of two different actions to be taken after examining the specified expression. There are two types of **if** statements.

Type 1 : **if** < Boolean expression > **then** < statement > ;



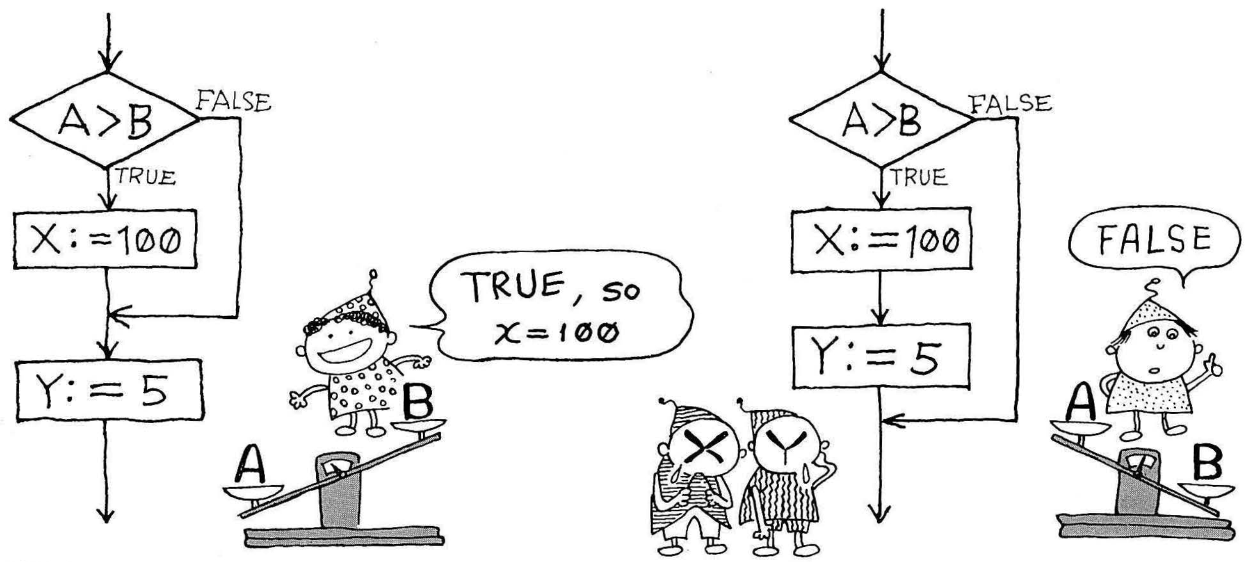
This type of the **if** statement executes the statement following **then** when the Boolean expression gives *true*, otherwise, the next statement is executed.

if A = 0 **then** A : = A+1 ;
- A = 0 — Boolean expression
- A : = A+1 ; — assignment statement

The assignment statement is executed when A is 0; otherwise the following statement is executed. This type of PASCAL **if** statement is the same as the BASIC IF statement.

Only one statement can be placed after **then**. If two or more statements are required, they must be grouped as a single compound statement.

Examine the difference between the following examples.
if A > B **then** X : = 100 ; Y : = 5 ;
if A > B **then begin** X : = 100 ; Y : = 5 **end** ;



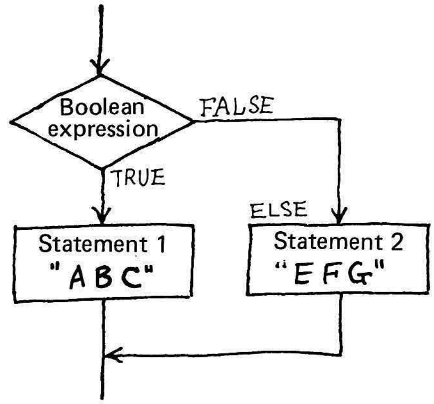
IF A>B THEN X:=100; Y:=5; IF A>B THEN BEGIN X:=100; Y:=5 END;

In the first example, the **if** statement ends with X : = 100 and Y : = 5 is executed separately. In the second example, the **if** statement consists of the whole line.

Type 2 : **if** < Expression > **then** < statement 1 > **else** < statement 2 > ;

This type of the **if** statement executes statement 1 when the Boolean expression gives *true*, otherwise it executes statement 2.

if *odd* (A) **then** *write* ("ABC") **else** *write* ("EFG") ;



When A is odd, the above statement executes *write* ("ABC"), then the statement following *write* ("EFG"). When A is even, it executes *write* ("EFG"), then following statement.

then and **else** can each be followed by only one statement. To specify multiple statements, combine them into a compound statement using **begin** and **end**.

An **if** statement may appear after **then** or **else**, but take note of the following.

if A>9 **then if** A<100 **then** Y : =Y+1 **else** X : =X+1 ;

The meaning of this statement differs according to whether **else** corresponds to the first **if** or the second **if**. In the statement above, **else** corresponds to the second **if** according to the rules of PASCAL (see Figure 5.1). To make **else** correspond to the first **if**, the statement must be written as follows. (See Figure 5.2).

if A>9 **then begin if** A<100 **then** Y := Y+1 **end else** X := X+1 ;

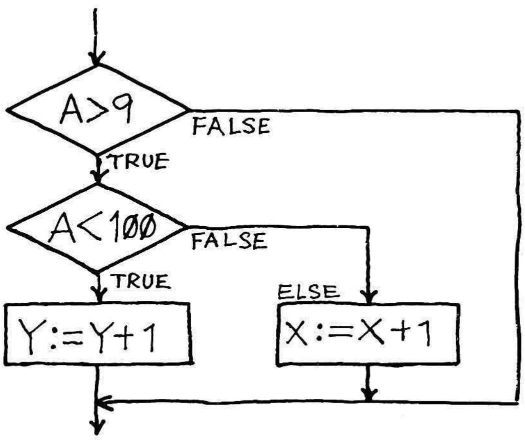


Figure 5.1

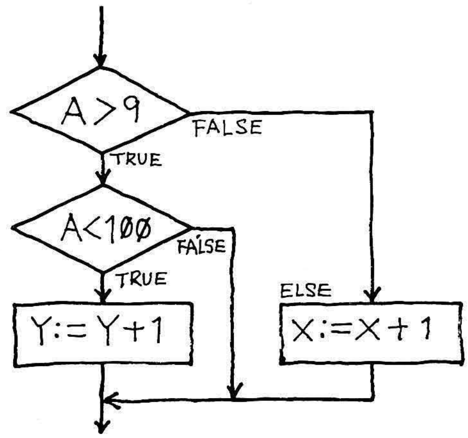
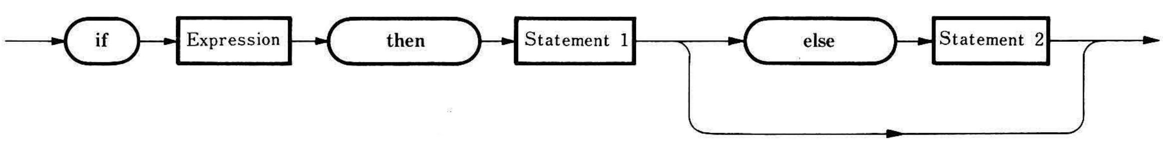


Figure 5.2

In Figure 5.1, X : = X+1 is executed when A is 100 or more and Y : = Y+1 is executed when A is 10 through 99. In Figure 5.2, X : = X+1 is executed when A is 9 or less and Y : = Y+1 is executed when A is 10 through 99.

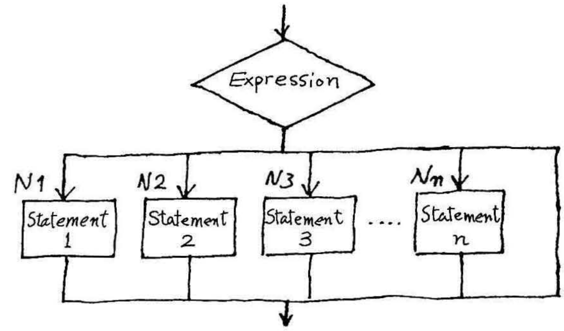
if statement



CASE statement (selection)

The **case** statement executes one of several different statements after examining the specified expression, which may be of any type.

```
case I of   1:  X := A+B ;
Expression  2:  X := A-B ;
            3:  X := A*B
end ;           Case labels
```



In the above example, when I is 2, X : = A-B is executed; when it is 3, X : = A*B is executed.

When the value of I is not specified (after **of**), the statement following the **case** statement is immediately executed. The constant values after **of** (which determine the statement to be executed) are called **case labels.**

```
case I of 4:   X := A*A ;
          5,6: X := A*A*A
end ;
```

In the above example, when the value of *integer* variable I is 4, X : =A *A is executed; when I is 5 or 6, X : = A* A *A is executed; otherwise, the statement following the **case** statement is executed.

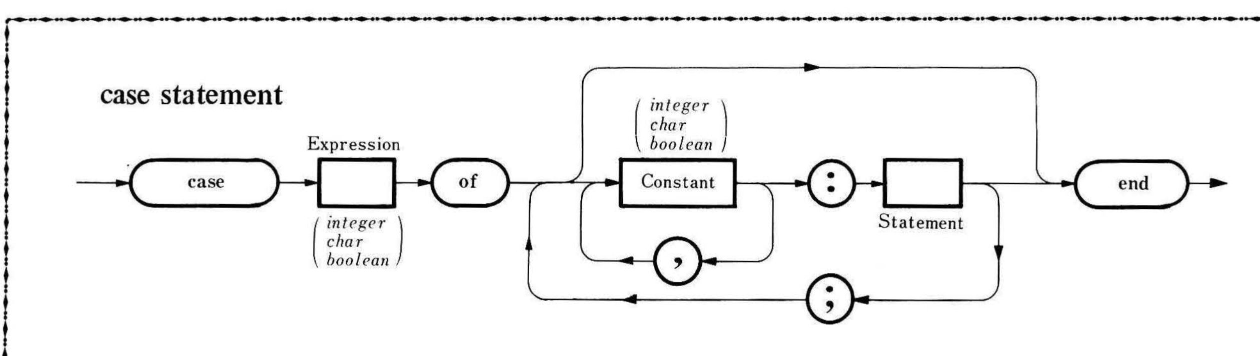
```
case CH of 'A': write ("CHARACTER CODE A IS 65") ;
           'B': write ("CHARACTER CODE B IS 66")
end ;
```

When the value of character variable CH is 'A', "CHARACTER CODE A IS 65" is displayed and when it is B, "CHARACTER CODE B IS 66" is displayed; otherwise the statement following the **case** statement is executed.

```
case X>=0.0 of true  : write (sqrt (X)),
               false : write ("IMPOSSIBLE TO CALCULATE")
end ;
```

When the value of *real* variable X is greater than or equal to zero, *write* (*sqrt* (X)) is executed; when it is negative, "IMPOSSIBLE TO CALCULATE" is displayed.

integer values which can be used as case labels range from 0 ~ ±32767. *char* values which may be used are those shown in the ASCII Code Table. Each case label used in a **case** statement must be unique.

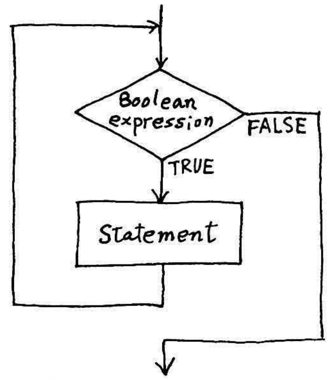


WHILE statement (repetition 1)

There are three means provided for repeating a statement or phase until a given condition is satisfied; these are the **while**, **repeat** and **for** statements.

The basic format of the **while** statement is as follows.

while < Boolean expression > **do** < statement > ;



While the Boolean expression is *true*, the specified statement is repeated. The statement will not be executed at all, however, if the initial state of the Boolean expression is *false*.

```
while A > 0 do A : = A-1 ;
```

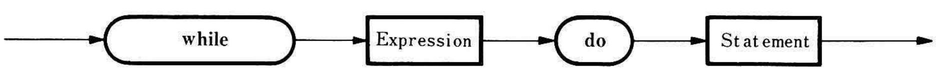
In the above example, A : = A-1 is executed when the value of variable A is greater than 0; this is repeated until A becomes 0, at which time the next statement is executed. When A is negative to start with, the specified statement is not executed at all.

To repeat several statements, group them as a compound statement using **begin** and **end**.

The following sample program gives the sum of integers 1 through 100.

```
var N , S : integer ;
   begin
      N : =0 ;
      S : =0 ;
      while N<100 do
         begin N : =N+1 ; S : =S+N end ;
      write ( " S= " , S : 4)
   end.
```

while statement

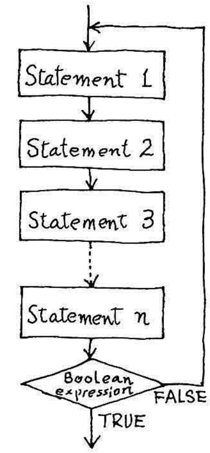


REPEAT statement (repetition 2)

The **repeat** statement executes a statement or a phase, then checks the value of the expression; if it is *true*, the next statement is executed, otherwise the statement or phase is repeated. Its basic format is as follows.

repeat < statement 1 > ; < statement 2 > ; . . . ; < statement n > **until** < Boolean expression > ;

```
0. var X , Y : real;
1.   begin
2.     X : =1.0 ;
3.      repeat
4.        Y : =sqrt (X) ;
5.        writeln ( " ROOT " , X : 3 , " = " , Y : 10) ;
6.        X : =Y+1.0
7.     until X=11.0
8.   end.
```

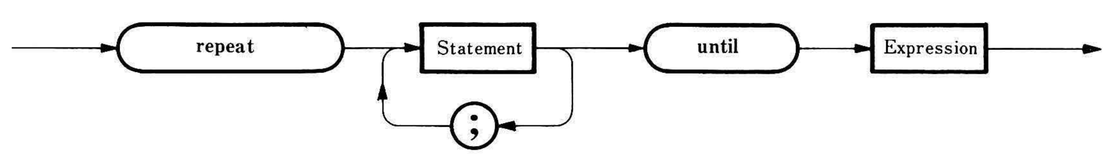


The above sample gives the square roots of the numbers 1 through 10. Since all variables are *real*, the constants must also be *real*. X : 3 and Y : 10 on line 5 indicate the display location, this will be explained in the explanation of the *write* statement.

There must be at least one statement between **repeat** and **until**; it is **not necessary** to group multiple statements using **begin** and **end**.

Note: The difference between the **repeat** statement and the **while** statement is that the specified statement(s) is always executed at least once with the **repeat** statement.

repeat statement



Writing PASCAL programs

In the various sample programs which have been described so far, you may have noticed the difference in the manner in which PASCAL and BASIC programs are written.

```
0. var R,AREA : real;
1. begin
2.   readln (R) ;
3.   while R<>0.0 do
4.     begin
5.       AREA : =3.14*R*R/4.0 ;
6.       writeln (AREA) ;
7.       readln (R)
8.     end
9. end.
```

```
0. VAR R, AREA : REAL;
1. BEGIN
2. READLN(R);
3. WHILE R<>0.0 DO
4. BEGIN
5. AREA: =3.14*R*R/4.0;
6. WRITELN(AREA);
7. READLN(R)
8. END
9. END.
```

The sample programs above are the same except for the style in which they are written. Execute them and note that they give the same results. The sample on the left is written so that the program structure is apparent. As shown above, indenting program phases appropriately makes it easier to read and understand the structure of a program.

One of the most convenient features of PASCAL is that **indentation** can be used in program coding. The number of spaces preceding each statement is not limited, but typically 2 spaces are used.

Each statement ends in a semicolon (;), and not with a carriage return code; therefore, statements could be written as shown below.

```
var   R,AREA : real;   ——→
```

```
var
  R,
  AREA :
  real;
begin
  :
```

Note that there is no semicolon (;) at the end of lines 7 and 8 in the sample program at the top of this page. This is because **end** serves to mark the end of statements in place of the semicolon (;).

Exercise: Write the following program using indentation. The key to doing this correctly is to determine which **if** corresponds to which **else**. Be sure to use semicolons (;) where needed.

```
VAR A, X : INTEGER
BEGIN READ (A)
IF A < 10 THEN X : = 1 ELSE IF A < 100 THEN X : = 2 ELSE IF A < 1000 THEN X : = 3 ELSE X : = 4
WRITE (X : 8) END
```

The solution is given on page 97.

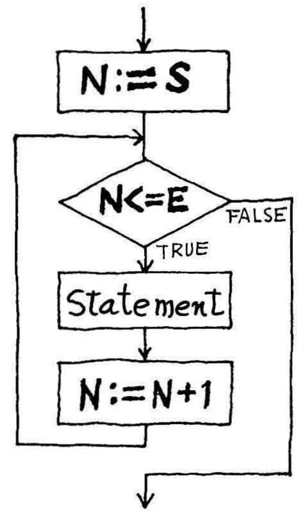
This program reads a positive value from the keyboard and displays "1" when the value read is one digit, "2" when two digits, "3" when 3 digits and "4" when 4 digits.

FOR statement (repetition 3)

This statement repeats a loop a specified number of times. It is similar to the FOR ~ NEXT statement of BASIC. There are two types of **for** statement provided in PASCAL.

Type 1 : **for** < control variable > : =< starting value > **to** < ending value > **do** < statement > ;

The control variable, starting value and ending value must be either *integer* values or *char* values. The control variable stores the starting value plus the number of repetitions performed. The starting value is first assigned to the variable and compared with the ending value. When it is less than or equal to the ending value, the statement following **do** is executed and the control variable is incremented by one. The next statement is executed when the control variable value becomes greater than the ending value.

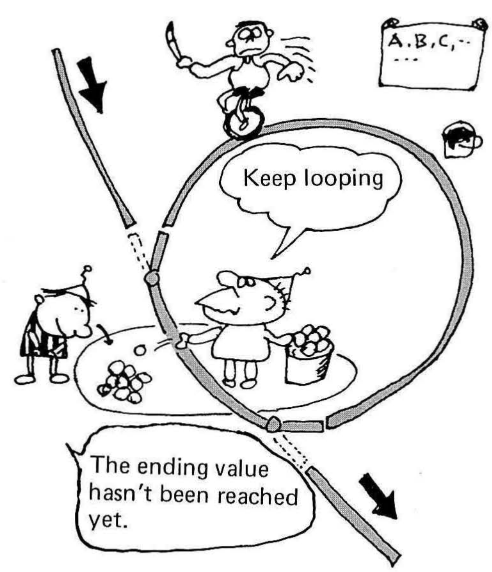


FOR N:=S TO E DO<Statement>

```
var N : integer;
    CH : char;
begin
   for N :=32 to 255 do
      begin
         CH : =chr (N) ;
         writeln ("CHARACTER FOR CODE ",N:3,
         " IS",CH:2)
      end
end.
```

The above sample program displays characters corresponding to ASCII codes 32 through 255.

```
var CODE : integer;
    CH : char;
begin
   for CH : ='A' to 'Z' do
      begin
         CODE : =ord (CH) ;
         writeln ("CHARACTER CODE FOR",CH:2,
         " IS",CODE:3)
      end
end.
```



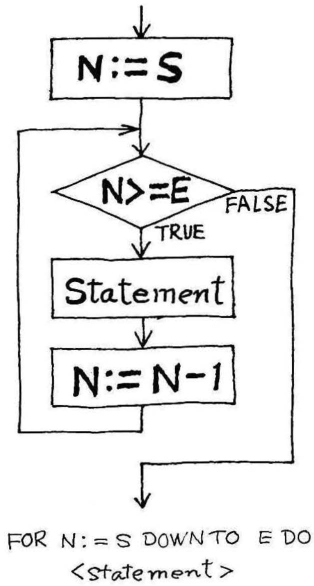
The above sample program displays the ASCII codes corresponding to characters A through Z. Note that the control variable, the starting value and the ending value are all *char* values. The character codes are displayed in the order in which letters are listed in the ASCII code table.

When it is necessary to execute several statements, group them as a compound statement and place them after **do**.

Type 2 : **for** <control variable> : = <starting value> **downto** <ending value> **do** <statement> ;

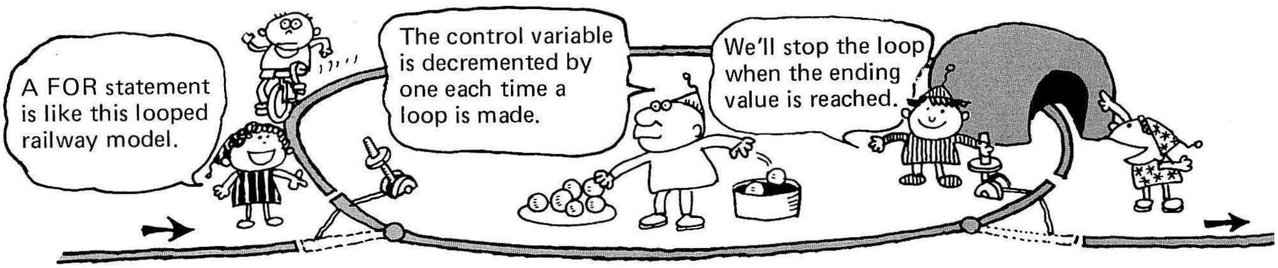
This type of **for** statement differs from type 1 in that the value of the control variable is decremented by one each time a loop is made. Otherwise it is the same as type 1.

```
var N : integer;
      CH : char;
begin
   for N : =32  downto  255  do
      begin
         CH : =chr (N) ;
         writeln ("CHARACTER FOR CODE",N:3,
         "IS",CH:2)
   end
end
```



The sample program shown above is a modification of that shown in the description of the type 1 **for** statement, with **to** replaced by **downto**. Nothing happens when this program is executed, because the starting value is not greater than the ending value.

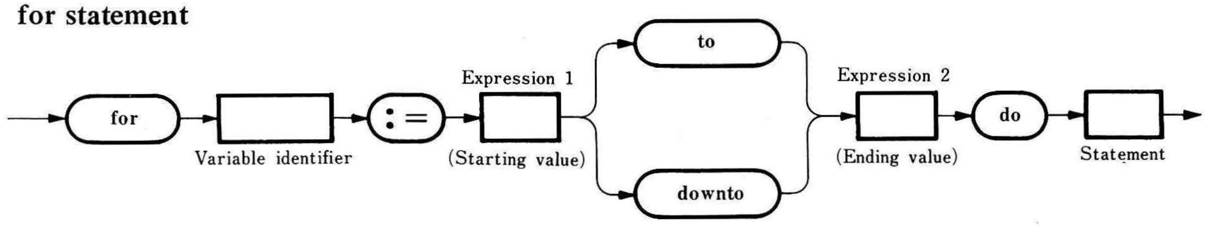
Replace 32 with 255 and vice versa and execute the program for the type 1 **for** statement.



A PASCAL program and a BASIC program are compared below. Notice that a FOR loop can include another FOR loop in both PASCAL and BASIC. But there is no limit on the number of nested levels of such loops in PASCAL.

PASCAL	BASIC
VAR X,Y: INTEGER;	10 FOR X=1 TO 9
BEGIN	20 FOR Y=1 TO 9
FOR X:=1 TO 9 DO	30 PRINT"X*Y=";X*Y
FOR Y:=1 TO 9 DO	40 NEXT Y
WRITELN("X*Y=",X*Y:2)	50 NEXT X
END.	

for statement



Procedure declaration and procedure (calling) statement

A procedure is a particular set of actions which may be used several times in a program. It corresponds to a subroutine in BASIC. A procedure must be declared in the procedure declaration section. There are two types of procedure declarations.

Type 1 : **procedure** <identifier> ; <compound statement> ;

The identifier corresponds to the subroutine name.

```
procedure SUM ;
   begin
     Z := X+Y     BLOCK
   end ;
```

Procedure SUM assigns the sum of X and Y to Z.
The following is a sample of a complete program which includes this procedure.

```
 0. var Z, X, Y : integer;            Variable declaration section
 1. procedure SUM ;
 2.    begin
 3.       Z := X+Y                    Procedure declaration section
 4.    end ;
 5. begin
 6.    readln (X) ;                   Executable statements
 7.    readln (Y) ;
 8.    SUM ; ........................ Procedure statement
 9.    writeln (Z)
10. end.
```

In the above sample, program execution starts at line 5.
Key-in data is assigned to X at line 6.
At line 7, other key-in data is assigned to Y.
Program control is transferred to procedure SUM at line 8 without any change in the values of X and Y. After the sum of X and Y has been assigned to Z by the procedure, program control is returned to line 9.
The value of Z is output at line 9.
The program terminates at line 10.

The last **end** in a program must always be followed by a period (.), not by a comma (,) or a semicolon (;).

Type 2 : **procedure** < identifier > (< formal parameter identifier > , . . . , < formal parameter identifier > : < type >) ; < variable declaration statement > ; < compound statement > ;

A procedure's action is based on the data assigned to its formal parameters when it is called. For type 1, values can be assigned only to the variables used within the procedure. For type 2, values can be assigned to any variables declared in the **var** declaration since the variables are assigned to the formal parameters specified for the procedure when each call is made.

```
 0. var X,Y,SUM,DIF : real
 1. procedure CALCULATION (A,B : real) ;
 2.    begin
 3.       SUM : =A+B ;
 4.       DIF : =A-B
 5.    end ;
 6. begin
 7.    readln (X) ;
 8.    readln (Y) ;
 9.    CALCULATION (X,Y) ;
10.    writeln ("X+Y= ",SUM) ;
11.    writeln ("X-Y= ",DIF) ;
12.    CALCULATION (SUM,DIF)
13.    writeln ("(X+Y)+(X-Y)= ",SUM) ;
14.    writeln ("(X+Y)-(X-Y)= ",DIF)
15. end.
```

Flow of program execution

1. Program execution starts at line 6.
2. Data values for X and Y are read from the keyboard at lines 7 and 8.
3. Procedure CALCULATION is called at line 9 with variables X and Y assigned to formal parameters A and B, respectively. ($X \rightarrow A$, $Y \rightarrow B$)
4. A+B and A-B are performed by procedure CALCULATION and the results are assigned to variables SUM and DIF, then program control is returned to line 10.
5. The results are displayed at lines 10 and 11.
6. Procedure CALCULATION is called again at line 12. At this time, variables SUM and DIF are assigned to formal parameters A and B, respectively. Calculations are performed and the results are assigned to variables SUM and DIF, respectively. Program control is then returned to line 13.
7. The results are displayed at lines 13 and 14.

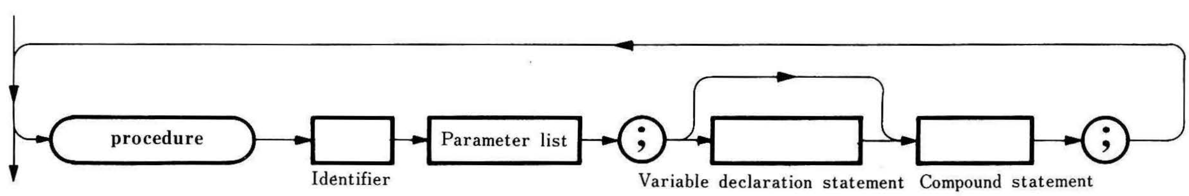
Let's review the meaning of the parameters. In the above program, A and B in line 1 are variables and are called formal parameters. Variables assigned to these formal parameters are called actual parameters.

It is not necessary to declare formal parameters. Identifiers of variables which are declared in the **var** declaration may be used as formal parameters. The number of formal parameters is not limited.

Note the following when using formal parameters.

1. The number of actual parameters used when a procedure is called must be the same as the number of formal parameters. For example, specifying CALCULATION (X) or CALCULATION (X, Y, Z) when calling the procedure declared by **CALCULATION** (X, Y : *real*) will result in an error.
2. The type of the actual parameters must be the same as the type of the formal parameters. In the above example, only *real* data can be assigned.
3. Formal parameters must be variables (expressions are not allowed).
   Thus, **procedure** (X+Y: *real*) is not a valid procedure declaration.
4. FILE identifiers cannot be used as formal parameter.

Procedure declaration



Function declaration and function designator

If the expression defined in a function includes parameters, the values of variables assigned to the parameters are used to perform the calculation. A function is different from a procedure in that the result of the calculation is assigned to a **"function identifier"**, rather than to a variable, then control is returned to the statement which designates the function.

A function must be defined in advance by a function declaration.

There are two types of function declaration.

Type 1 : function < function identifier > : < result type > ; < variable declaration statement > ;
< compound statement > ;

Type 2 : function < function identifier > (< formal parameter > , . . . , < formal parameter > : < type >) :
< result type > ; < variable declaration statement > ; < compound statement > ;

The following example defines a function which gives the area of a triangle, S = ah/2.

```
function AREA (A,H : real) : real;
   begin
      AREA : =A*H/2.0
   end ;
```

AREA is both the function identifier and a variable. The following sample program includes this function declaration.

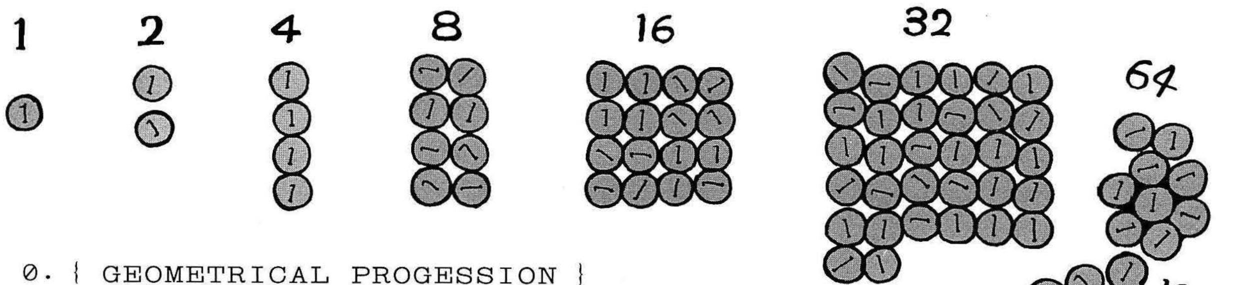
```
 0. var X,Y : real:                                                      Variable declaration
 1. function AREA (A,H : real) : real;                                   }
 2.    begin                                                             } Function declaration
 3.       AREA : =A*H/2.0                                                }
 4.    end ;                                                             }
 5. begin                                                                }
 6.    write ("BASE  A=") ;                                              }
 7.    readln (X) ;           Reads the value of the base length         }
 8.    write ("HIGHT  H=") ;                                             } Executable statements
 9.    readln (Y) ;           Reads the value of the height.             }
10.    write (AREA  =") ;     } Displays the result.                     }
11.    write (AREA (X,Y))     }                                          }
12. end.     Function designator
```

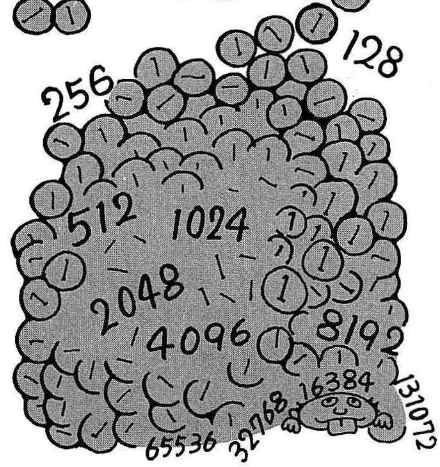
Details of program execution at line 11 are as follows. This statement is an instruction which displays the value of variable AREA. However, since this variable has not yet been calculated, program control is passed to the function AREA with variables X and Y (containing values entered from the keyboard) assigned to A and H. A*H/2.0 is calculated in the function declaration block and the result is assigned to function identifier AREA. Program control is then returned to the statement on line 11 and the value of AREA is displayed.

Sample program

Assume that you want to accumulate coins in geometrical progression; for example, 1 coin on the first day, 2 coins on the second day, 4 coins on the third day and so on.

The number of coins which will have accumulated after a certain number of days can be calculated with the following sample program.

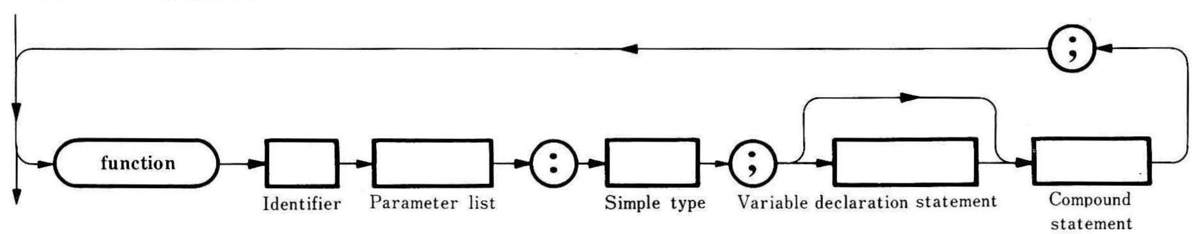





```
{ GEOMETRICAL PROGESSION }
var M,SUM : real;
    N,X : integer;
function TOTAL (DAY : integer) : real;
   begin
     if DAY=1 then TOTAL : =1.0
       else begin
              SUM : =1.0 ;
              M : =1.0 ;
              for N : =2 to DAY do
                begin
                  M : =2.0*M ;
                  SUM : =SUM+M
                end ;
              TOTAL : =SUM
           end
   end ;
begin
   write ("◎") ;
   X : =1 ;
   while X<>0 do
     begin
       writeln ( ) ;
       write ("⇩⇩ ◆ HOW MANY DAYS YOU ACCUMULATE COINS.") ;
       readln (X) ;
       writeln ("⇩ ** TOTAL IS",TOTAL (X) : 8, "COINS")
     end
end.
```

This calculation could be performed with a procedure instead of a function, but the number of variables would have to be increased because no value can be assigned to a procedure identifier. Try coding a program which uses a procedure to obtain the same result.

Function declaration



Global variable and local variable

A procedure declaration or function declaration can declare variable which are valid only in the declaration block. Such variables are called **local** variable. Arrays can also be declared as local variables. Variables which are declared in the variable declaration block are called **global** variables. Global variables are valid throughout the program.

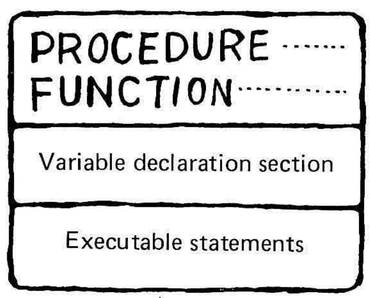
```
 0. var A : integer ;  .............. Declares global variable A. (A is valid throughout the pro-
 1. procedure TAB (X : integer) ;     gram.)
 2.    var N : integer ; ............ Declares local variable N. (N is valid only within the procedure de-
 3.       begin                       claration block for TAB.)
 4.          for N : =1 to X do
 5.             write ( " ⇒ " )
 6.       end ;
 7. begin
 8.    write ( " ©THE NUMBER OF TABS ARE " ) ;
 9.    readln (A) ;
10.    TAB (A) ;
11.    write ( " ABC " )
12. end.
```

No function corresponding to TAB (X) in BASIC is provided in PASCAL. The above sample program provides a similar function using a procedure declaration. When the program is executed, it asks the operator for the number of tabs. Key in an appropriate *integer* number and check the position of "ABC" on the display screen.

Variable N is declared at line 2. This variable is valid only within the declaration block for procedure TAB. Therefore, it cannot be used within another parts of the program. Further, no value can be externally assigned to it.

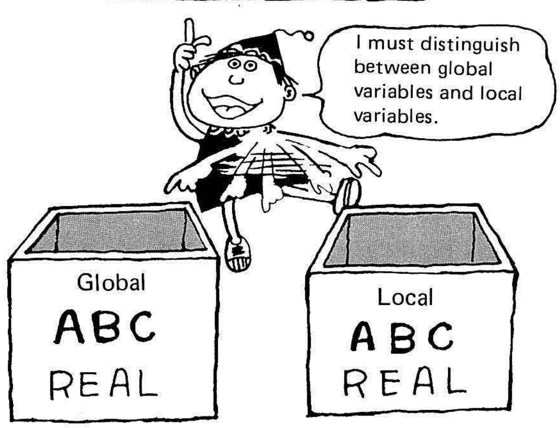
Parameter X is automatically defined as a local variable.

The structure of a procedure declaration or function declaration block is as shown at right. It is similar to the structure of PASCAL programs in general.



Modify the sample program "GEOMETRICAL PROGRESSION" shown on the preceding page as follows and execute it.

```
{ GEOMETRICAL PROGRESSION }
var SUM : real;
       X : integer:
funcnction TOTAL (DAY : integer) : real;
   var M : real; N : integer;
      begin
         :
```



The following sample program will clarify the difference betwen global variables and local variables.

```
0. var N : char; ········· Declares global variable N.
1. procedure PRINT;
2.   var N : char; ········· Declares local variable N.
3.     begin
4.       N := 'B' ;
5.       writeln ("LOCAL VARIABLE N IS",N:2)
6.     end;
7. begin
8.   N := 'A' ;
9.   writeln ("A IS FIRST ASSIGNED TO GLOBAL VARIABLE N.");
10.  PRINT;
11.  writeln ("CHECK THE CONTENTS OF GLOBAL VARIABLE N.");
12.  writeln ("GLOBAL VARIABLE N IS",N:2)
13. end.
```

This program uses the same identifier for both global and local variables. Program execution proceeds as follows.

(1) Line 7 is the beginning of the executable statement section.
(2) Character A is assigned to global variable **N** at line 8.
(3) A message is output at line 9.
(4) Procedure PRINT is called at line 10.
(5) In the procedure declaration block, character B is assigned to local variable **N** at line 4.
(6) The contents of **local variable N** are displayed at line 5 and program control is returned to line 11.
(7) A message is displayed at line 11.
(8) The contents of **global variable N** are displayed at line 12.

Character A, which was first assigned to global variable N, remains unchanged after program execution. Local variable N is valid only within the procedure PRINT.

Local variables are:
(1) Variables which are declared in procedure and function declarations, or
(2) Formal parameters of procedure and function declarations.

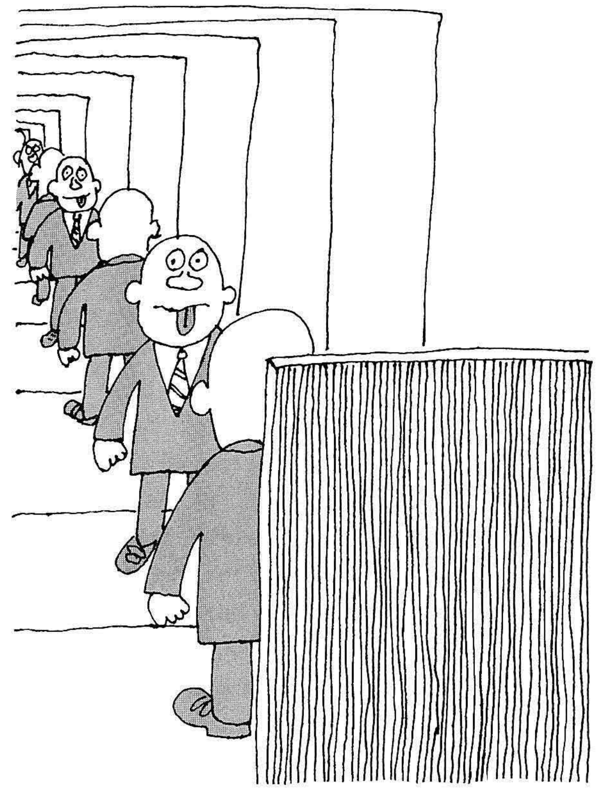
Global variables are variables which are declared at the beginning of programs by **var** declarations.

Local variables may be defined as files.

Recursion

A procedure (or function) may call itself. Such cases are called recursion. In BASIC, recursion is what occurs when a subroutine calls itself.

The following sample program gives the sum of integers 1 through N.



```
var K : integer;
function SUM (N : integer) : integer;
begin
   if N=1 then SUM : =1
      else SUM : =SUM (N-1) + N
end ;                    Recursive call
begin
   readln (K) ;
   writeln ( " SUM= ", SUM (K) : 6)
end
```

In the above sample program, the function SUM calls itself with N-1 assigned to the parameter. The structure of this program is difficult to understand, and it is difficult to write a clear flow chart. However, the program structure can be clarified with a diagram called an NS chart.

With BASIC, recursive calls are generally impossible except in the case shown below.

Recursion with BASIC (precisely speaking, this is not really recursion for the reason described on page 10.)

```
10   INPUT " N = " ;N
20   GOSUB 100
30   PRINT "END"
40   STOP
100  PRINT "N =  " ;N
110  IF N=0 THEN RETURN
120  N=N-1
130  GOSUB 100
140  RETURN
```

The subroutine itself is called at line 130; therefore, this may be regarded as a recursive call. However, when the value of N is large, the maximum number of subroutine levels is exceeded.

With PASCAL, there is no limit on the number of recursive calls which may be made other than the limit imposed by the useable memory capacity. Therefore, care is required when using recursion.

Most programs which use recursion could be written without it. Recursion does not reduce execution time or the amount of memory required by the program.

It is sometimes better not to use recursion. Whether or not recursion is used must be determined on a case-by-case basis.

However, use of recursion often makes the program structure easier to understand. The following programs both give the factorial of N; the first uses recursion and the second does not.

```
var X : integer ;
function FACTORIAL (N : integer) : integer ;
   begin
      if N=0 then FACTORIAL : =1
         else FACTORIAL : =N*FACTORIAL (N-1)
   end ;                                   Recursive call
begin
   write ( " © " ) ;
   for X : =0 to 7 do
      begin
         writeln (X : 1 , " ! " ,FACTORIAL (X) : 5) ;
         writeln ( )
      end
end
```

```
var X : integer ;
function FACTORIAL (N : integer) : integer ;
   var A , B : integer ;
   begin
      A : =1 ;
      B : =0 ;
      while B<N do
         begin
            B : =B+1 ;
            A : =A*B
         end ;
      FACTORIAL : =A
   end ;
begin
   write ( " © " ) ;
   for X : =0 to 7 do
      begin
         writeln (X ; 1 , " ! " ,FACTORIAL (X) : 5) ;
         writeln ( )
      end
end
```

WRITE statement

The *write* statement is used to display a calculation result or a message on the CRT screen, to print it out on the printer or to write data on cassette tape. It corresponds to the PRINT statement in BASIC.

There are several forms of *write* statements as shown below.

Type 1 : for display of a character string on the CRT

> *write* ("< character string > ") ;
> *writeln* ("< character string > ") ;

These statements display the character strings enclosed in double quotation marks (") on the CRT screen. The *write* statement does not make a carriage return after it has displayed the character string, but the *writeln* statement does. Ⓒ Ⓗ ⇧ ⇩ ⇦ ⇨ may be enclosed in double quotes in the same manner as in BASIC.

Type 2 : for printing a character string on the printer

> *pwrite* (" < character string > ") ;
> *pwriteln* (" < character string > ") ;

The only difference between this form and type 1 is that the character string is output to the printer.

Type 3 : output of the value of an expression

> *write* (<expression 1 > : <expression 2 > : <expression 3 > , . . .) ;
> *writeln* (<expression 1 > : <expression 2 > : <expression 3 > , . .) ;

These statements display the value of expression 1 so that the least significant digit is displayed in the position which is a certain number of spaces to the right of the current cursor position. This number is determined by expression 2. Expression 3 is valid only when expression 1 is *real*, it specifies the number of decimal places. Expression 1 may be any type of expression other than *boolean*, expressions 2 and 3 must be *integer* expressions.

write ('A' : 8) : — 1 2 3 4 5 6 7 8
Character A is displayed at the 8th position to the left of the current cursor position. — □□□□□□□$\boxed{A}$

write ('A' : 3, 'B' : 2, 'C' : 4) ; — 1 2 3 4 5 6 7 8 9 — □□$\boxed{A}$□$\boxed{B}$□□□$\boxed{C}$

write ('A') ; — 1 2 3 4 .... 12 13 14 15
The default value of expression 2 is 15. — □□□□······□□□$\boxed{A}$

Assuming that 1.2345 is assigned to *real* variable X. — 1 2 3 4 5 6 7 8
write (X : 8) ; — □□$\boxed{1}$$\boxed{.}$$\boxed{2}$$\boxed{3}$$\boxed{4}$$\boxed{5}$
write (X : 5) ; An error results since the number of digit of the contents of X is 6.
write (X : 5 : 2) ; — 1 2 3 4 5
The contents of X are displayed down to the 2nd decimal place. — □$\boxed{1}$$\boxed{.}$$\boxed{2}$$\boxed{3}$
write ("ABC", 'X' : 3) ; — $\boxed{A}$$\boxed{B}$$\boxed{C}$□□$\boxed{X}$
write ('X' : 3, "ABC") ; — □□$\boxed{X}$$\boxed{A}$$\boxed{B}$$\boxed{C}$

The above rules also applies to the *writeln*, *pwrite* and *pwriteln* statements.

Assume that 2, 3 and 8 are assigned to *integer* variables X, Y and Z, respectively.

```
                                         1 2 3 4 5 6
write (X+Y : Z-X) ; ----------------------□□□□□5
expression 1     expression 2
```

The above example is the same as *write* (5 : 6) since expression 1 gives 5 and expression 2 gives 6. In *write* (X, Y, Z) all variables are treated as non-*file* variables when X is not a *file* variable, even though the others are. As shown above, *file* declaration is checked only for the first variable; other variables are assumed to be the same type as the first variable. This is also true for type 2 *read* statements.

writeln ( ) performs a carriage return. *write* ( ) does not result in an error, but no action is performed, (other than to reduce the running speed).

The following statements dispaly values of array elements which are not declared as *file*.

```
write (DATA [15] : 5) ;          write (DATA [X-Y] : A+B) ;
Array identifier   Expression 2     Expression 1    Expression 2
           Expression 4
      Expression 1
```

(The second statement displays the same data as the first one when X is 20, Y is 5, A is 2 and B is 3.)

Expression 3 can be specified when the array is *real* as follows.

```
write (DATA [15] : 5 : 3) ;
  Expression 1
          Expression 2   Expression 3
```

Type 4 : Output of data which is declared as *file*

write (< identifier > , < identifier > , . . . , < identifier >) ;

For example, the following statement records integers 1 through 10 on the cassette tape as the data file "10 INTEGERS".

fname ("10 INTEGERS") ; **for** N: = 1 **to** 10 **do** *write* (N) ; *close*

In this case, variable N must be declared as *file* in advance; otherwise, the data will be displayed on the CRT screen. The variable identifier specified in a *write* statement must also be specified in a *read* statement when the recorded data is to be read. For example, to read data recorded in the above example, use.

for M : = 1 **to** 10 **do** **begin** *read* (N) ; . . . . ; **end**

The following statement cannot read the data because the read data is assigned to control variable N.

for N : = 1 **to** 10 **do** **begin** *read* (N) ; . . . . ; **end**

write (X, Y, Z) results in an error when X is declared as *file* and Y and Z are not. *file* declaration is checked only for the first identifier. Variables declared as *file*, it may be *boolean* variables.

Type 5 : Output of data of arrays which are declared as *file*

write (< array identifier > [ ], < array identifier > [ ], ..., < array identifier > [ ]) ;

This statement saves all array element data in the cassette tape file when the array are declared as *file*. No character may be enclosed in [ ]. Any data type may be used.

write (RESULT [ ])

The above example saves all array element values from the array RESULT in the cassette tape file. The number of dimensions of the array is not limited. It is not possible to save part of an array by specifying RESULT [5].

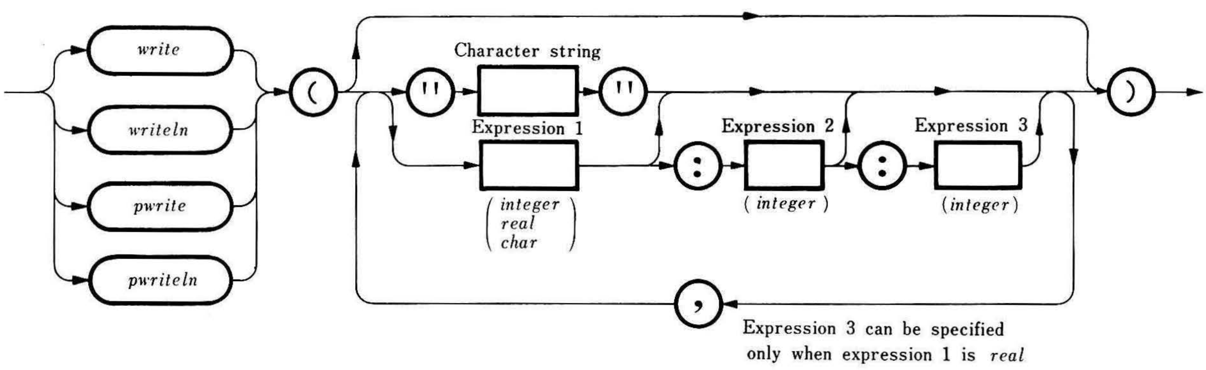
fname ("ABC") ;
write (RESULT [ ], DAY [ ]) ;
close ;

When the above statements are executed, all data from arrays RESULT and DAY are saved in the cassette tape file with the file name ABC assigned. In this case, executions *read* (DAY [ ]) with file name ABC specified results in an error. *read* (RESULT [ ], DAY [ ]) must be used.

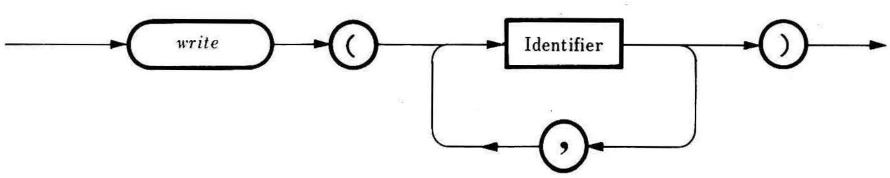
To store arrays RESULT and DAY in different files (or different tapes), the tape deck must be stopped after the array RESULT has been stored. Therefore, the program is written as shown below.

fname ("ABC") ;
write (RESULT [ ]) ;
close ;
read (A) ; .... Program execution is stopped until a key is pressed (the tape deck is also stopped).
fname ("DEF") ;
write (DAY [ ]) ;
close ;

write statement



write statement (type 4 and 5)



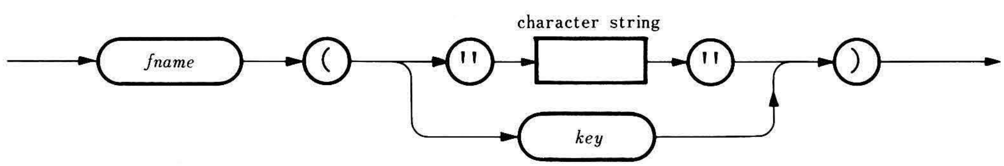
The following sample program stores integers 1 through 5 in a cassette file, then reads them from the file.

```
0. var X,M : integer;
1.     N : FILE OF integer;
2. procedure PUTDATA;
3.    begin fname ("5 integers");
4.       for N:=1 to 5 do write (N) ; close
5.    end;
6. procedure GETDATA;
7.    begin fname ("5 integers");
8.       for M:=1 to 5 do
9.          begin read (N) ; X:=N ; write (X:4) end ; close
10.   end;
11. begin
12.   writeln ("© DATA WILL BE STORED IN THE CASSETTE TAPE
    FILE.");
13.   PUTDATA;
14.   writeln ("⇩ DATA HAS BEEN STORED IN THE CASSETTE TAPE
    FILE.");
15.   writeln ("⇩ PRESS ONE OF KEYS 0 THROUGH 9 AFTER REWIND
    HAS BEEN COMPLETED.");
16.   readln (X);
17.   writeln ("⇩ DATA WILL BE READ FROM THE CASSETTE TAPE
    FILE.");
18.   GETDATA;
19.   writeln ();
20.   write ("END")
21. end.
```

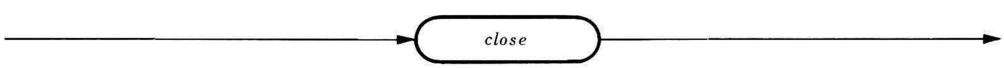
The *read* statement on line 9 is explained in the next section. X : = N on line 9 is required because N is declared as *file* and it cannot bespecified in a *write* statement for screen display.

Note: No expression can be specified in the parentheses of *write* statement of type 4 or type 5.

fname statement



close statement



READ statement

The *read* statement reads data from the keyboard or the cassette tape. It corresponds to the INPUT statement of BASIC.

Type 1 : Reading the values of variables which are not declared as *file*

read (<identifier>, <identifier>, . . . ,
<identifier>) ;

When this statement is executed, ? is displayed to request that data be keyed in when the identifiers are not declared as *file*. Key in data and press the [CR] key and the keyed data is read and displayed. No carriage return is performed when the *read* statement is executed.

read (X, Y, Z) ;

When the above statement is executed, the system displays ? and waits for input of the data to be assigned to X. After the first data has been input and the [CR] key has been pressed, the system displays ? after the data just entered to wait for data input to Y. After the data for Y has been entered, the system displays ? to wait for data for Z. After all data has been read, the system goes onto the next statement.

Data can be keyed in another way, as shown in Note 5 below. Attention must be paid to the cursor position when many items of data are keyed in.

Type 2 : Reading the values of variables which are not declared as *file.*

readln (<identifier>, <identifier>, ... , <identifier>) ;

This statement is the same as type 1 except than a carriage return is carried out after the last data has been read.

Notes:
1) No *boolean* variables can be specified in a *read* statement of type 1 or type 2.
2) Only one character can be read when a variable is *char* variable.
3) No expression can be specified in parentheses.
4) For *read* (X, Y, Z), non of the variables will be handled as *file* variables if X is not a file variable. *file* declaration is checked only for the first variable.
5) For *read* (X, Y, Z), data can be keyed in two ways. For example, to assign 5 to X, 6 to Y and 7 to Z, key in [5]↲ [6]↲ [7]↲ or [5][,][6][,][7]↲ .

Type 3 : Reading variables which are declared as *file*

read (< identifier > , < identifier > , ... , < identifier > ,

When the variables are declared as *file*, the system automatically reads the file data. The file must be opened, and the file name must be declared by *fname* statement in advance. The data read is not displayed on the CRT screen. After reading has been finished, the cassette tape stops and the system executes the next statement.

Executing *read* (X, Y, Z) results in an error when X is declared as *file* and Y and Z are not. *file* declaration is checked only for the first variable.

readln statements are not used for variables declared as *file*.

Type 4 : Reading array variables which are declared as *file*.

read (< array identifier > [ ] , < array identifier > [ ] , ... , < array identifier > [ ]) ;

When the variables to be read are array variables which are declared as *file*, this statement reads the values of all elements of the array. No character can be specified within [ ].

Data can be read even if the array identifier or the number of dimensions of the array is different from that specified when data are saved, if the total number of array elements and the data type are the same as those stored in the file. See the example on page 36.

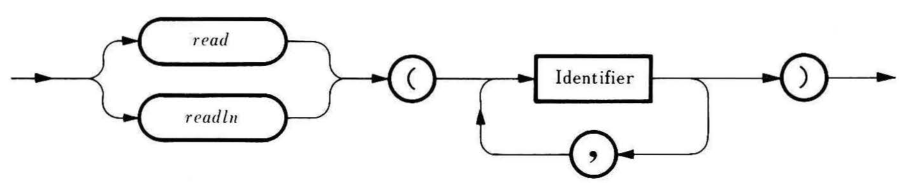
All array identifiers specified must be declared as *file*; otherwise, an error results.

Notes: 1. *boolean* variables can be used when they are declared as *file*.

2. No statement or expression can be specified within parentheses.

read (X+Y) ;	Incorrect because an expression is used in parentheses.
read (Z := X−Y) ;	Incorrect because a statement is used in parentheses.
read (X, Y, Z) ;	An error results when X is declared as *file* but Y and Z are not. When Y and Z are declared as *file* but X is not, no error results but Y and Z are treated as if they are not declared as *file*.

read statement (type 1 and 2)



Graphic control statements

The MZ-80B personal computer can be used for display of high-density graphics by installing an optional graphic RAM card. Graphic control with PASCAL is almost the same as with BASIC. The PASCAL graphic control statements and functions are listed below with the corresponding BASIC control statements and functions for comparison.

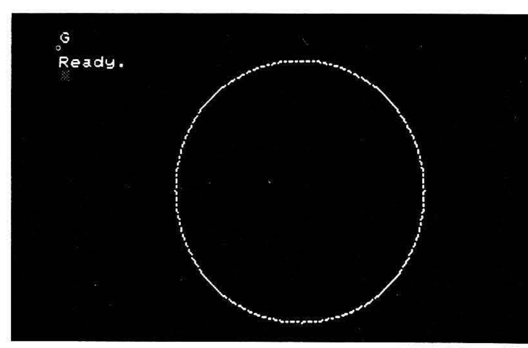
PASCAL graphic control statements	BASIC graphic control statements
graph (< I, a, O, b, C, F >)	GRAPH <Ia, Ob, C, F>
gset (x, y)	SET x, y
grset (x, y)	RESET x, y
line $(x_1, y_1, x_2, y_2 <, x_3, y_3, \ldots, x_n, y_n >)$	LINE $x_1, y_1, x_2, y_2 <, x_3, y_3 \ldots, x_n, y_n >$
bline $(x_1, y_1, x_2, y_2 <, x_3, y_3, \ldots, x_n, y_n >)$	BLINE $x_1, y_1, x_2, y_2 <, x_3, y_3 \ldots, x_n, y_n >$
position (x, y)	POSITION x, y
pattern (x_1, <"character string" $\mid$ character expression>)	PATTERN $x_1, x_1\$ <, x_2, x_2\$ \ldots x_n, x_n\$ >$

PASCAL graphic control functions	BASIC graphic control functions
point (x, y)	POINT (x, y)
posh	POSH
posv	POSV

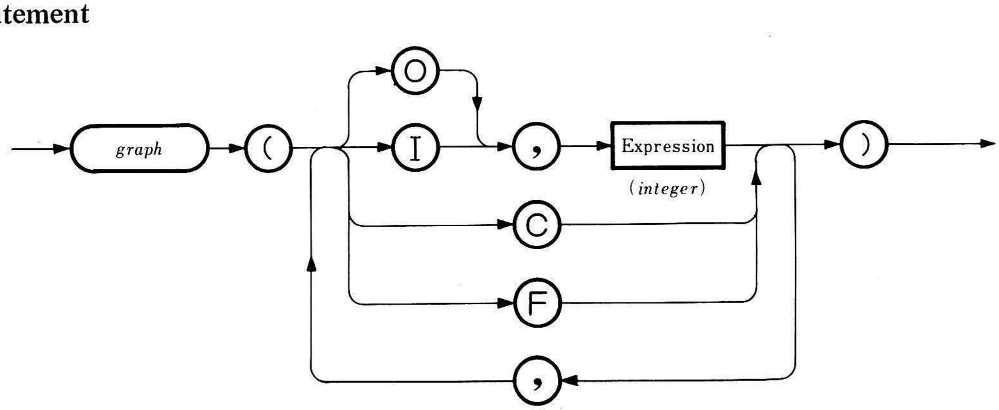
Lets use the *gset* statement to draw a circle on the screen with a radius of 80 whose center is at (160, 100). We can do this by rotating a radius vector of 80 through 360° (1° at a time) to set dots. The coordinates of the radius vector can be computed using the SIN and COS functions with respect to the angle. Note that the parameters and the results of the SIN and COS functions are of the *real* type, whereas the operands of the *gset* statement must be of the *integer* type. Consequently, it is necessary to convert data types when passing arguments between graphic control statements.

A programming example and the results of its execution are shown below.

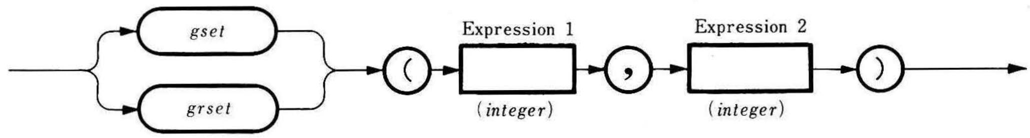
```
var X,Y,TH : integer; DK : real;
begin
   graph (I,1,C,0,1);
   for TH:=0 to 360 do
     begin
        DK:=float (TH) * 3.1415927/180.0;
        X:=trunc (cos (DK) * 80.0) +160;
        Y:=trunc (sin (DK) * 80.0) +100;
        gset (X,Y)
     end
end.
```



graph statement

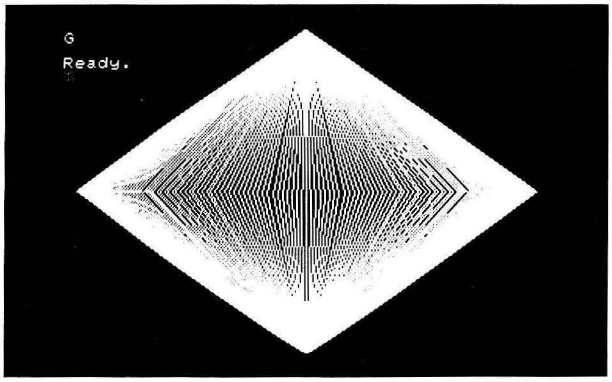


gset/grset statement

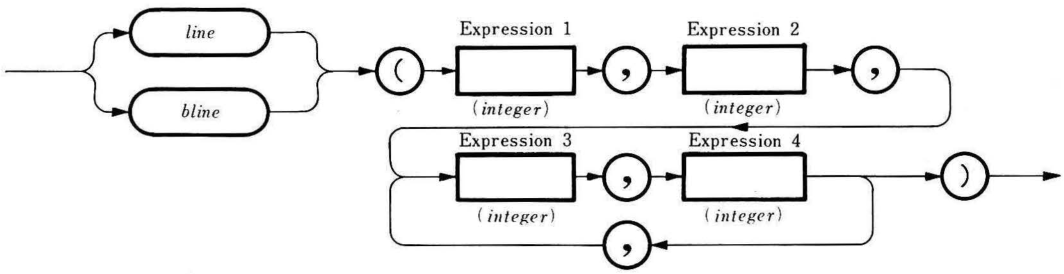


Let's draw a diamond on the screen using the *line* statement. Note that the coordinate data specified in the operand field of the *line* (or *bline*) statement must also be of the *integer* type.

```
var A : integer ;
begin
   graph (I,1,C,O,1) ;
   for A : =0 to 150 do
   line (160,0,trunc (cos (float (A)
         * 3.1415927/150.0) +160,
         100,160,200) ;
end.
```

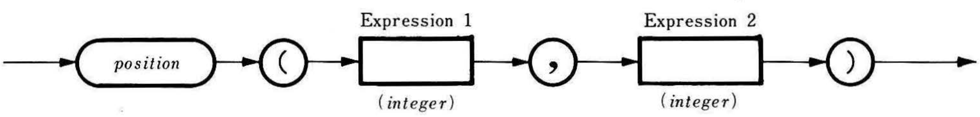


line/bline statement

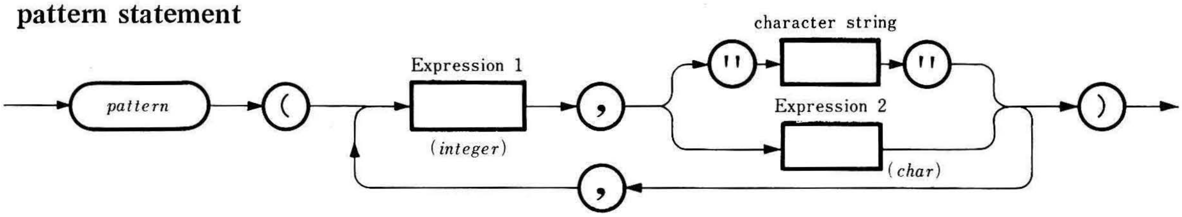


You can use the *position* and *pattern* statements of PASCAL in the same manner as in BASIC. Note that pattern data specified in the operand field of the *pattern* statement must be a character string enclosed in double quotation marks or character type data.

position statement



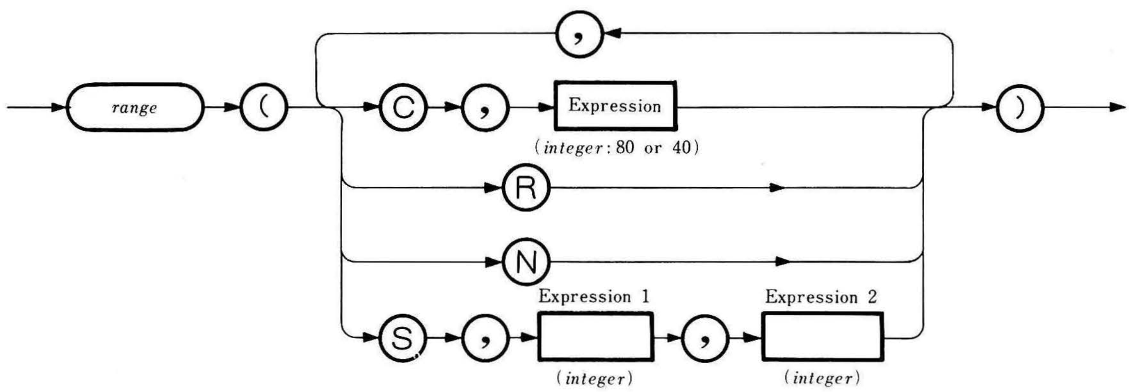
pattern statement



Character display control statements

The *range* statement fixes the scrolling area of the character display screen, changes the character display mode between 80 characters/line and 40 characters/line, or character and graphic display mode between reverse mode and normal mode.

range statement



The operand of the *range* statement determines which of three functions shown below are activated.

- **Fixing the scrolling area**

  range (. . . S, *ls*, *le* . . .) . . . . The top line refers to line 0 of the display and the bottom line to line 24. ls and le fix the scrolling area.
  Hence, $0 \leqq ls < le \leqq 24$
  This area, however, must cover at least three lines.

- **Changing the character display mode**

  range (. . . C, 80 . . .) . . . . This sets the character display mode to "80 characters/line".
  range (. . . C, 40 . . .) . . . . This sets the character display mode to "40 characters/line".

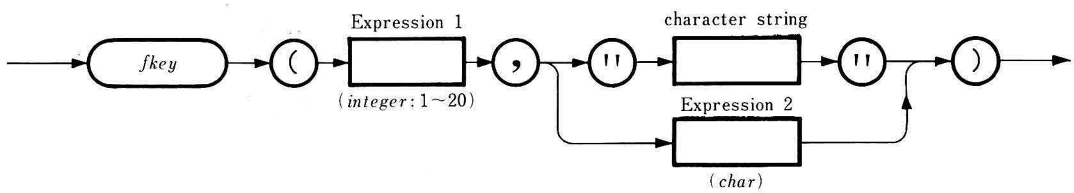
- **Changing the character and graphic display mode**

  range (. . . R . . .) . . . . This sets the character and graphic display mode to reverse mode.
  range (. . . N . . .) . . . . This sets the character and graphic display mode to normal mode.

Function key control and printer control statements

The *fkey* statement corresponds to the DEF FKEY statement in the BASIC language and can be used to define any of twenty functions of the ten definable function keys.

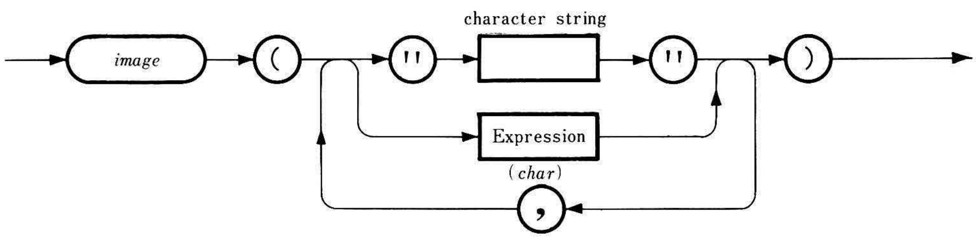
fkey statement



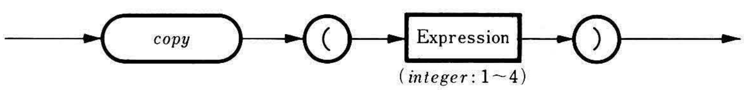
A number from 1 through 20 is assign to function number, provided that a number from 11 through 20 can be used to define functions in SHIFT state.

The *image* statement causes the printer to draw a desired dot pattern according to the operating mode (image mode 1 or 2), and the *copy* statement causes to copy an entire frame of data displayed on the computer screen. These are corresponding to the BASIC statements IMAGE/P and COPY/P respectively.

image statement



copy statement



Example: *image* ("π UU π") . . . . Draws the dot pattern "目" on the line printer.
image ("π", "UU", "π") . . . . . Draws the three dot patterns " | ", "≡" and " | " extending over three lines on the line printer.
copy (1) . . . . Causes the printer to copy the character display.

CALL statement

This statement calls a subroutine coded in machine language; it corresponds to the USR statement of BASIC.

call (< expression >) < variable identifier 1 >, < variable identifier 2 > ;

This statement stores the value of variable 1 in the HL register and the value of variable 2 in the DE register, then jumps to the address indicated by the expression. A return is made by the RET instruction. The expression and variables must be *integer* type in decimal notation. The variables may be declared as *file*.

When 32000 is assigned to variable X, 752 to variable Y, 4608 to variable B and 100 to variable C, executing

call (X+Y) B, C ;

causes 4608 ($1200 in hexadecimal) to be stored in the HL register and 100 to be stored in the DE register and program control to be transferred to address 32752 ($7FF0) (i.e. X+Y).

The user must be familiar with the machine language to use this statement. Careless use of this statement may result in destruction of the program.

When program control is returned to the PASCAL program from the subroutine, the HL and DE register contents at return are assigned to variables 1 and 2, respectively. Therefore, if the values of variables 1 and 2 at the time the subroutine is called are necessary, save them with PUSH instructions at the beginning of the subroutine and restore them with POP instructions at the end of the subroutine as shown below.

The stack pointer contents must be the same before and after execution of the subroutine. Thus, the general structure of a subroutine is as follows.

```
LD (nn'), SP        nn'  :  A hexadecimal address, e.g. $FFF0
LD SP, mm'          mm'  :  Specified by the user
PUSH HL
PUSH DE
   :      ┐
   :      │
   :      ├ User subroutine
   :      │
   :      ┘
POP DE
POP HL
LD SP, (nn')
RET
```

A negative decimal value must be specified in the statement when a hexadecimal data or address value is equal to or greater than $8000. The method for converting a hexadecimal value into the respective decimal is explained below.

1. Converting hexadecimal values up to $7FFF to decimal

```
        $1000     $0100    $0010
          ↓         ↓        ↓
$7FFF  = 4096 x 7 + 256 x 15 + 16 x 15 + 15 = 32767

              $7FFF

$4A0B  = 4096 x 4 + 256 x 10 + 16 x 0  + 11 = 18955

$0250  = 4096 x 0 + 256 x 2  + 16 x 5  + 0  =   592
```

2. Converting hexadecimal values greater than $8001 to decimal

(1) Convert the hexadecimal value to binary.
(2) Invert each bit.
(3) Add 1 to the result.
(4) Convert the resulting binary value to hexadecimal, then convert it to decimal with a – sign affixed.
(5) $8000 is to be handled as –32767–1. This is a special case.
(6) Care must be taken with operations X after X:=–32767–1 has been executed.

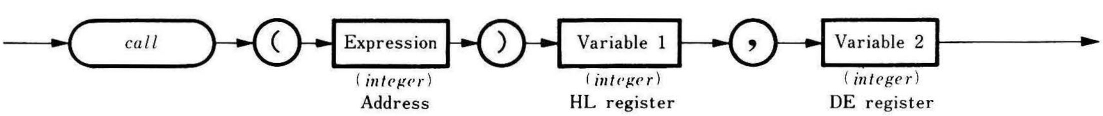
Example:
Converting $8F56 into decimal.

```
The binary expression of $8F56 is ............ 1000 1111 0101 0110 ........... $8F56
Invert each bit .............................. 0111 0000 1010 1001 ........... $70A9
Add 1 ........................................ 0111 0000 1010 1010 ........... $70AA
The decimal expression of $70AA is ........... 4096x7 + 256x0 + 16x10 + 10 = 28842
Affix the – sign ............................. –28842
```

To call a subroutine starting at address $8F56, execute *call* (–28842).
A program which performs the above conversion is shown in the appendix.

$FFFF is –1 in decimal notation.
$8001 is –32767 in decimal notation.
$8000 is –32767–1 in decimal notation.

call statement



COUT statement

This statement displays a character at the current cursor position according to the ASCII code table.

cout (< expression >) ; This statement displays *char* data indicated by the expression at the current cursor position. The cursor position is not changed by execution of this statement.

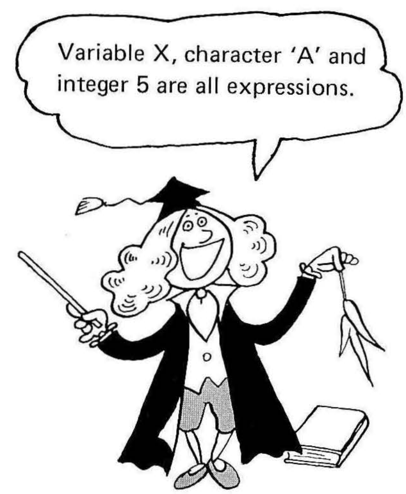
PATT := *cin* ; . . . . Assigns the character in the cursor position to the variable PATT.

⋮

cout (PATT) ; . . . . The character assigned to PATT by *cin* is displayed at the cursor position.

cout (CHR (X)) displays # when X is 35 because ASCII code 35 corresponds to character #.
Expressions specified in *cout* statements may be declared as *file*.

> Expression
> Many representations (< expression >) have been used thus far in this manual. < expression > includes not only such as X+Y and A−B but also variables and constants without sign. See the syntax diagram for expressions in chapter 7.



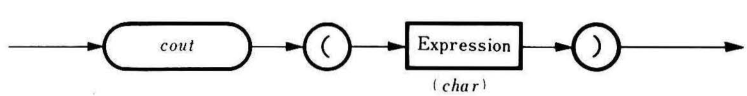
A : = X+Y is not an expression, but is a statement.

X+Y, A>B and *sqrt* (X) are expressions.

5 and 'A' are expressions.

It is a good practice to check the syntax diagrams when in doubt.

cout statement



POKE statement

The *poke* statement writes specified data in memory. It corresponds to the POKE statement of BASIC.

poke (<expression 1>, < expression 2 >);

This statement stores the code data given by expression 1 in the address indicated by expression 2. Expression 1 must be of type *char* and expression 2 of type *integer*. Both may be declared as *file*.

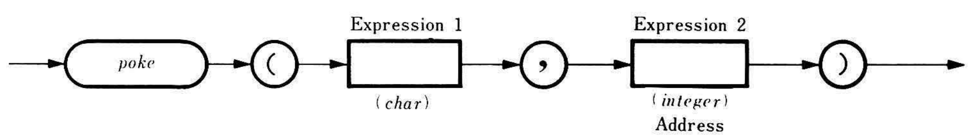
poke (*chr* (X), Y) ;

Assume that 65 is assigned to X and 32752 to Y. This statement then stores 65 ($41) in address 32752 ($7FF0). X is an *integer* variable, but *chr* (X) is *char*. This statement is equivalent to *poke* ('A', Y) in this case, since the ASCII code for the character A is 65.

The integer given by expression 1 must be within the range 0-255 because one exceeding 255 cannot be stored in one byte. The value of expression 2 must be negative when an address higher than $8000 is specified.

Data may not be written in the PASCAL interpreter area.

poke statement



OUTPUT statement

The *output* statement outputs data to the specified port. With this statement, peripheral devices can be controlled with a PASCAL program.

output (< expression 1 >, < expression 2 >) ;

This statement outputs data given by expression 1 to the port address indicated by expression 2. When this statement is executed, the data given by expression 1 is loaded into the A register (accumulator) and the address given by expression 2 is loaded into the BC register, then the following machine language instruction is executed.

OUT (C), A $ED79

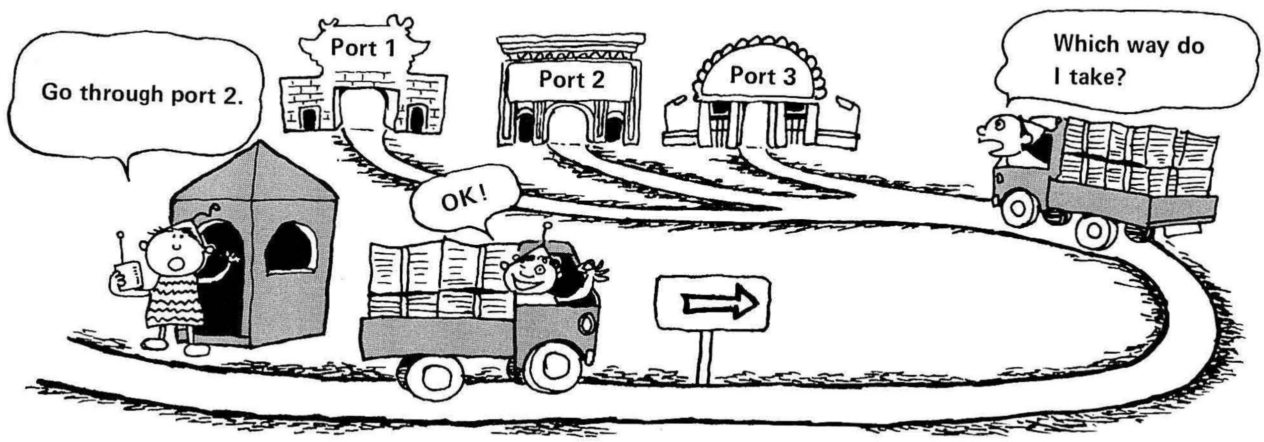
Expression 1 must be a *char* expression and expression 2 an *integer* expression. Both may be declared as *file*. The data code is in accordance with the ASCII code table.

Care must be taken with the value of expression 2 since port addressing is performed using the C register contents. For example, the following two statements specify the same port.

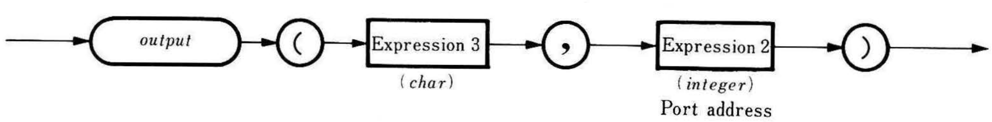
output (*chr* (X), 255) ;	255 is $00FF in hexadecimal.
output (*chr* (X), 4351) ;	4351 is $10FF in hexadecimal.

As shown above, the lower byte of the hexadecimal data is used to specify the port address. Therefore, no problem occurs when the value given by expression 2 is within the range 0-255.

To input data from a port, the *input* function is used.



output statement



EMPTY statement

An empty statement is one in which nothing is written.
See the following statement.

```
if  A=0  then
    else  B : = true ;
```

There is no statement after **then**, but an empty statement is executed.

```
if  A=0  then  B : = false
    else;
```

The above includes an empty statement after **else**. Thus, an empty statement can replace any statement in the syntax diagrams.

```
if  then  X : =1 ;
```

This statement results in an error because an expression, not statement, must be placed between **if** and **then**.
The following statements are correct, although they are not generally used.

1. **begin**
   end.

2. **begin** A : =X+Y ; **end;** Normally, this is written as **begin** A : = X+Y **end;**.

3. A : = X ; ; ; ;

Use of this type of statement is not recommended since they waste memory and make execution speed longer.

Statements and functions

A statement is a unit of program execution. **A function is not a statement**, but is included in a statement. Take note of and learn the following statements and functions in particular.

Statement	Function
output	*input*
poke	*peek*
cout	*cin*
	key

Any function can be a part of an expression but no statement can.

write ("DATA=", *cout* (X)); This statement is incorrect because a *cout* statement is used instead of an expression.

write ("DATA="); *cout* (X); Correct.

write (*peek* (X+Y)); Correct.

Exercise:

Find all errors in the following program and describe the reasons.

```
(1) while ord (key)=0 do key;
(2) if X<>0 then peek (25302);
(3) var A,B : real;
    function SUM (X,Y; real) : real;
      begin
        SUM : =X+Y
      end;
    begin
      readln (A,B) :
      S U M  (A,B) ;
      write (SUM (A,B) )
    end.
```

(Solution is given on page 90.)

Exercise

The following sample program gives the solution of a quadratic equation. This program executes only once. Rewrite it so that it can loop any number of times and execution can be ended at any time.

```
 0. { QUADRATIC EQUATION }
 1. var A,B,C,D : real;
 2. function JUDGE (E,F,G : real) : boolean;
 3.    begin
 4.       D:=F*F-4.0*E*G; { D=B*B-4*A*C }
 5.       if D>=0.0 then JUDGE := true
 6.          else JUDGE := false
 7.    end;
 8. procedure ROOT (K : boolean);
 9.    var SROOT,ROOT1,ROOT2,ROOT3,ROOT4 : real;
10.    begin
11.       case K of
12.          true : begin
13.                    SROOT := sqrt (D);
14.                    ROOT1 := (-B+SROOT) / (2.0* A);
15.                    ROOT2 := (-B-SROOT) / (2.0* A);
16.                    writeln ("THE ROOT OF 1 IS",ROOT1);
17.                    writeln ("THE ROOT OF 2 IS",ROOT2);
18.                 end;
19.          false : begin
20.                    ROOT3 := -B/(2.0*A);
21.                    ROOT4 := sqrt (-D) /(2.0*A);
22                     writeln ("THE ROOT OF 1 IS",ROOT3:12," + ",
                                                  ROOT4:12," I ");
23                     writeln ("THE ROOT OF 2 IS",ROOT3 12," - ",
                                                ROOT4 12," I ")
24.                 end
25.       end
26.    end;
27. begin
28.    writeln ("©⇩⇩⇨⇨⇨AX⇧2⇩ + BX + C = 0");
29.    write ("⇩⇩      A IS");
30.    readln (A);
31.    write ("⇩      B IS");
32.    readln (B);
33.    write ("⇩      C IS");
34.    readln (C);
35.    write ("⇩⇩");
36.    ROOT (JUDGE (A,B,C))
37. end.
```

Line 36 designates function JUDGE with the values of A, B and C read at lines 30, 32 and 34 be parameters, then calls procedure ROOT with the resultant data of function JUDGE be a parameter.

MUSIC statement and TEMPO statement

These statements enable the computer to play music. The *tempo* statement specifies the tempo and the *music* statement specifies notes to be played and plays it.

***tempo* statement** *tempo* (< expression >) ;

The expression is of the *integer* type and its result must be in the range from 1 through 7.

tempo (1) ; The slowest tempo (Lento, Adagio)

tempo (4) ; Medium tempo (Moderato): 4 times faster than *tempo* (1)

tempo (7) ; The fastest tempo (Molto Allegro, Presto): 7 times faster than *tempo* (1)

The *music* statement is executed as moderato (*tempo* (4)) when no *tempo* statement is specified initially.

***music* statement** *music* (< "character string" > | < *char* type expression >) ;

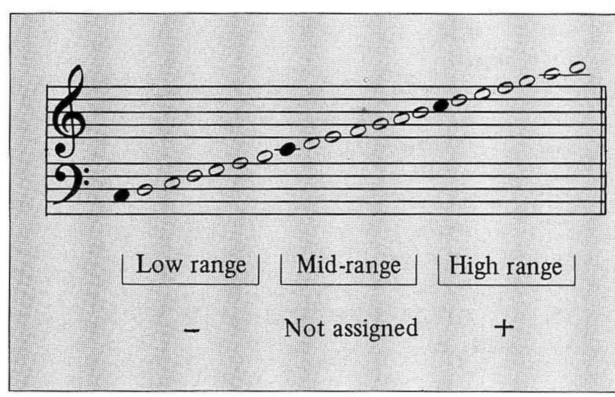
The *music* statement plays music according to the specified character string or *char* type expression at a tempo specified by the *tempo* statement.

The following indicates how the melody or sound effect converted into string data.
Musical notes are assigned according to pitch (octave and scale) and duration.

Octave assignment: – +

The sound range covers **three octaves** as shown at right. The black points indicate C notes, and the three C notes are separated by octave assignments as follows;

Low C . . . . – C
Middle C C
High C . . . . + C

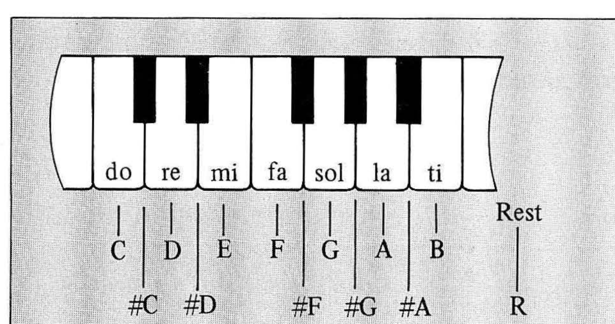


Note specification: C, D, E, F, G, A, B, # and R

C, D, E, F, G, A, B and # are used for note specification.

The relationship between the notes and these characters is shown at right. The # symbol is used for semitone assignment.

Rests (no sound) are assigned with **R**.



Duration specification:

This specification determines the duration of a note whose pitch has already been assigned. Note durations from thirty-second to whole are specified with numbers from **0 to 9**. The duration of rests (R) is also specified in this manner.

32nd rest	16th rest	Dot 16th rest	8th rest	Dot 8th rest	Quarter rest	Dot quarter rest	Half rest	Dot half rest	Whole rest
32nd note	16th note	Dot 16th note	8th note	Dot 8th note	Quarter note	Dot quarter note	Half note	Dot half note	Whole note
0	1	2	3	4	5	6	7	8	9

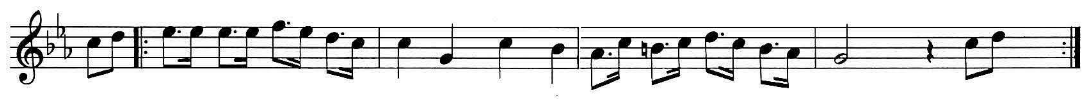
When notes of identical duration are repeated, duration specifications for the second and following notes may be omitted. If no duration specification is made, program execution is carried out with quarter notes (duration 5) initially.

Volume control

The sound output volume cannot be controlled by means of the program, but it can be controlled by the volume control provided on the rear of the cabinet.

Example:

The beginning of "Girl" by the Beatles is played by the following program.



```
0 . var A : integer ;
1 . begin
2 .     tempo (5) ;
3 .     A : =1 :
4 .     while A<>0 do
5 .         begin
6 .             music (" +C3+D+ #D4+ #D1+ #D4+ #D1+F4+ #D1+D4+C1+C5G+C#A") ;
7 .             music (" #G4+C1B4+C1+D4+C1B4 #G1G7R5 ")
8 .         end
9 . end.
```

This program repeats play. Line 7 can be rewritten using *char* expressions as follows.

7. *music* ('#', 'G', '4', ' + ', 'C', ...

Variables may be used in the above statement, and character strings and *char* type expressions can be mixed.

COMMENT statement

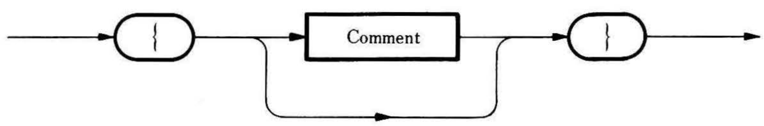
The comment statement is a non-executable statement which makes it easy to review the program list. It corresponds to the REM statement of BASIC.

{ character string }

Any number of comment may be used in any part of a program. However, frequent use of comment statements makes the running speed slower and requires a greater amount of memory. A comment statement can not be specified within another statement.

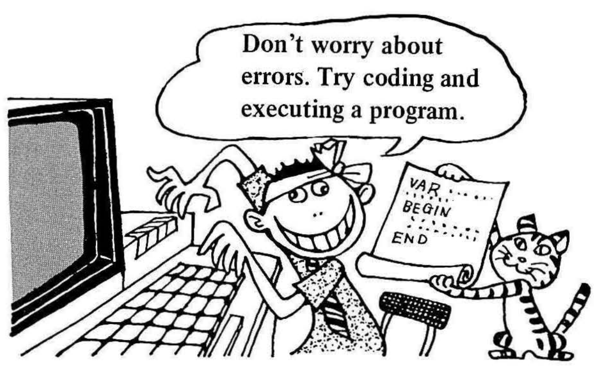
```
{ PUZZLE }
{    1981. 7.15   }
var  A,B,XMAX : integer;
     DATA : char:
procedure  UP (N : integer) ; {   CHARACTER  UP   }
   begin
     :
```

Comment statement



This concludes our explanation of the rules of the PASCAL. The many new statements and unique programming procedure will require some practice to gain familiarity with this new language.

Code and execute many programs, and you will become skillful in PASCAL programming.



Solution of exercise on page 86:

(1) A statement should follow **do**, but the function *key* does.

(2) A statement should follow **then**, but the function *peek* (25302) does.

(3) A statement should be placed on line 7, but the function SUM (A, B) is.

Chapter 6

Programming

Programming

Now that you are familiar with the rules of PASCAL, you are ready to try writing programs. The question lies in what approach to take. With BASIC, you can start keying in a program as soon as you have conceived it; this is possible because detailed sections can be developed as subroutines and linked to the main program with GOSUB as the need arises. This is not the case with PASCAL.

The fact that PASCAL does not include an equivalent of the GOTO statement means that the structural sequence of a PASCAL program must be well defined in advance. A natural result is that PASCAL programs are very clear and have a structure which is self apparent. Thus, learning to write programs with PASCAL requires developing a method of approach which will cause you to change your idea of programming in other languages as well. This is the main reason PASCAL is referred to as an educational programming language.

As was explained in chapter 1 of this manual, PASCAL programs are made by means of structured programming.

1. Make an outline of the process to be used in solving the problem and divide it into independent subprocesses. This is equivalent to writing subroutines. Parts which cannot be separated as subprocesses are left for inclusion in the main routine.

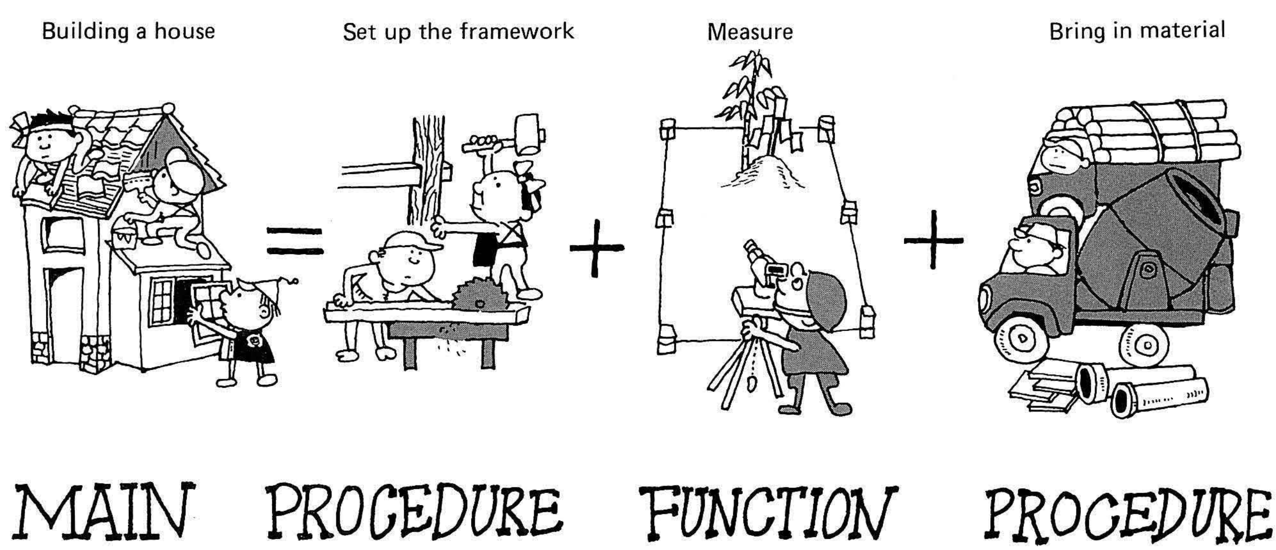
2. Each subprocess constitutes a procedure or function. Name (identify) the procedures and functions in any order.

3. Code PASCAL entries for one of the procedures or functions. It may be useful to break the subprocess into smaller components for convenience in coding. Declare variables which are used only within one coded block as local variables.

4. When the first block has been completed, go on the next one. You may consider incorporating previously coded blocks into a single one at this time.

5. Assign an identifier to each global variable used in coded blocks; make a list of all identifiers for global variables.

6. When all blocks have been completed, combine them into one body.



7. Insert a comment statement at the beginning of the program to identify it.

8. Next, declare global variables which are included in the list.

9. Now write declarations for procedures and functions which have been coded. The order in which these are arranged can be independent of the order of execution. Arrange them so that the overall program structure is readily apparent.

10. Executable statements come last. The order of these statements is extremely important since they determine the sequence of execution of the program. Arrange executable statements following the declaration section, starting them with **begin** and ending with **end**. Processes which cannot be broken down during step 1 are coded in this section.

11. Enter the program and run it to check for errors. If any error messages are output, correct the program according to the messages. Care must be taken when corrections are made because they may have an influence on other parts of the program.

> **Note :** A certain area in the memory is reserved for a variable or array when it is decoded. Indeterminate data is stored in this area at the time of declaration.
> For example, when
>
> ```
> var A : integer ;
> begin
>     write (A)
> end.
> ```
>
> is executed, some number will be output. This may be cause problems when an array is declared and data is assigned to only part of its elements. Thus, an array should always be initialized after declaration.

Indentation

It is recommended that statements be indented as described on page 59. This not only makes the program easy to read but also helps prevent errors when the program is entered. Indentation does not require additional memory space. The number of spaces preceding each statement is not limited, but generally two spaces are used for each statement level.

Thus, **end** is indented the same number of spaces as **begin**, and **until** the same number of spaces as **repeat. else** can be indented the same number of spaces as **if** or two spaces more than **if**,

```
begin
  repeat
    read (A);
    if A<>Ø then write (X)
      else write (Y)
  until A=1ØØ
end
```

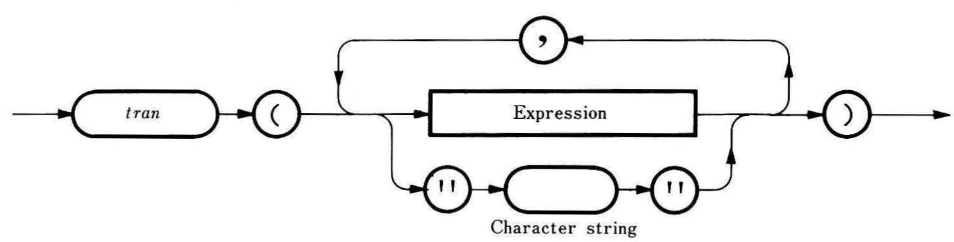
Link with color control system

Load the SB-3000 series cartridge in the cassette deck and turn on the power (start IPL). Then, load PASCAL interpreter SB-4515 series into memory.

Three statements and one function are provided for controlling the color control system. These are briefly outlined below; for details, refer to the Color Control Manual.

1. TRAN

This statement transfers a graphic command to the color display terminal.



Example 1 :

The following program displays the character string "SHARP" in red on a green background formed of 256 x 192 dots.

```
0. begin
1.    tran ("M, 0", "B, 2", "C, 1") ;
2.    tran ("SF, 127, 95, 0, SHARP")
3. end.
```

Lines 2 and 3 can be placed on one line. For the format of the character string in quotation marks, refer to the Color Control Manual.

Line 1 can be rewritten to include expressions which result in *char* data as follows.

tran ('M', ',', '0', *chr* (13), 'B', ',', '2', *chr* (13), 'C', ',', '1', *chr* (13)) ;

Example 2 :

The following program allows the color of the background and the characters to be specified from the keyboard.

```
0.{ COLOR CONTROL }
1.var BACKGND,CHARCOL,CR : char;
2.procedure COLCONT (X,A : char) ;
3.    begin
4.       tran ('B',',',X,CR) ;
5.       tran ('C',',',A,CR) ;
6.       tran ("SF,50,95,0,SHARP")
7.    end ;
8.begin
9.    CR : =chr (13) ;
10.   write ("©⇩⇩⇩⇨⇨* SPECIFY BACKGROUND COLOR[0--7] ") ;
11.   readln (BACKGND) ;
12.   while (BACKGND<'0') or (BACKGND>'7') do readln (BACKGND) ;
13.   write ("⇨⇨SPECIFY CHARACTER COLOR [0--7] ") ; ]
14.   readln (CHARCOL) ;
15.   while (CHARCOL<'0') or (CHARCOL>'7') do readln (CHA   OL) ;
16.   tran ("M,1") ;
17.   COLCONT (BACKGND,CHARCOL)
18.end.
```

Look at lines 4 and 5. These statements use expressions which result in *char* data and are concluded with carriage returns. Statements including such expressions must always be concluded with carriage returns.

2. REQTR

This function obtains 1 byte of data from the terminal.
No parameters are used and the result is *char* type data.

Example:

X : = *reqtr*

This function obtains 1 byte of data from the terminal and assigns it to *char* variable X. To convert the data to *integer* data, use Y : = *ord* (*reqtr*).

3. SYRET

This statement resets the color control display terminal and makes a cold system start.

4. SYRET2

This statement resets the color display terminal and causes the system to wait for entry of a monitor command (DU·A).

There is no statement corresponding to OTBIN in BASIC which transfers 1 byte of hexadecimal data to color display terminal, since this can be done using *tran.*

For example,

tran (*chr* (62), *chr* (13))

transfers hexadecimal data $3E to the terminal device, where 62 is the decimal value of $3E and 13 that of the carriage return. This statement can be rewritten as follows.

tran ('>', *chr* (13))

where > indicates ASCII code 62.

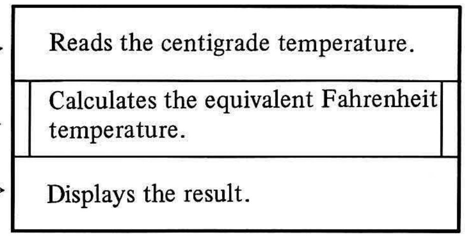
NS chart

Flowcharts are not used to represent the structure of PASCAL programs because they are not suitable for representing the structure of such programs. Instead, NS (Nassi Shneider) charts are used to portray the structure of PASCAL programs. NS charts are convenient for checking the flow of very complex programs. It is strongly recommended that you become familiar with use of these charts.

1. Compound Statement

Consider the compound statement shown below.

```
begin
    readln (TEMPC) ;
    TEMPF ; ………… Calls procedure TEMPF.
    writeln (TEMPF)
end .
```



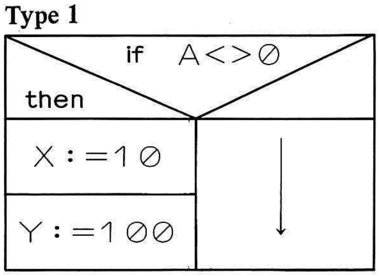
Draw a rectangle and divide it into sections corresponding to the program steps shown above. Write the first program step executed in the top section, the second program step executed in the second section and so on.

In the above example, the double lines at the ends of the center section indicate that a procedure or function is called. This is equivalent to representation of a subroutine in a flowchart.

2. IF Statement

Two types of NS charts are used for **if** statements since this statement is used in two forms.

Type 1

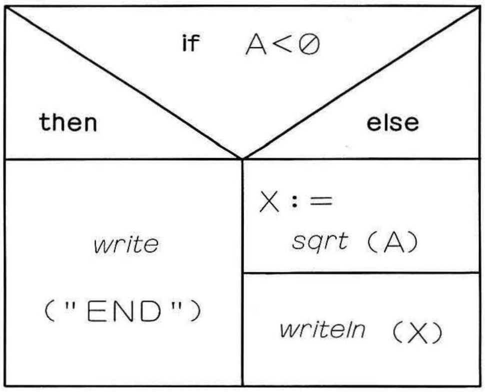



```
if A<> 0 then
   begin
      X ; =10 ;
      Y : =100
   end ;
```

The conditional expression is written in the inverted triangle. Statements to be executed when the condition is satisfied are written on the left side and the arrow on the right side indicates that the statements are to be bypassed when the condition is not satisfied.

Only one statement may be written in each section.

Type 2

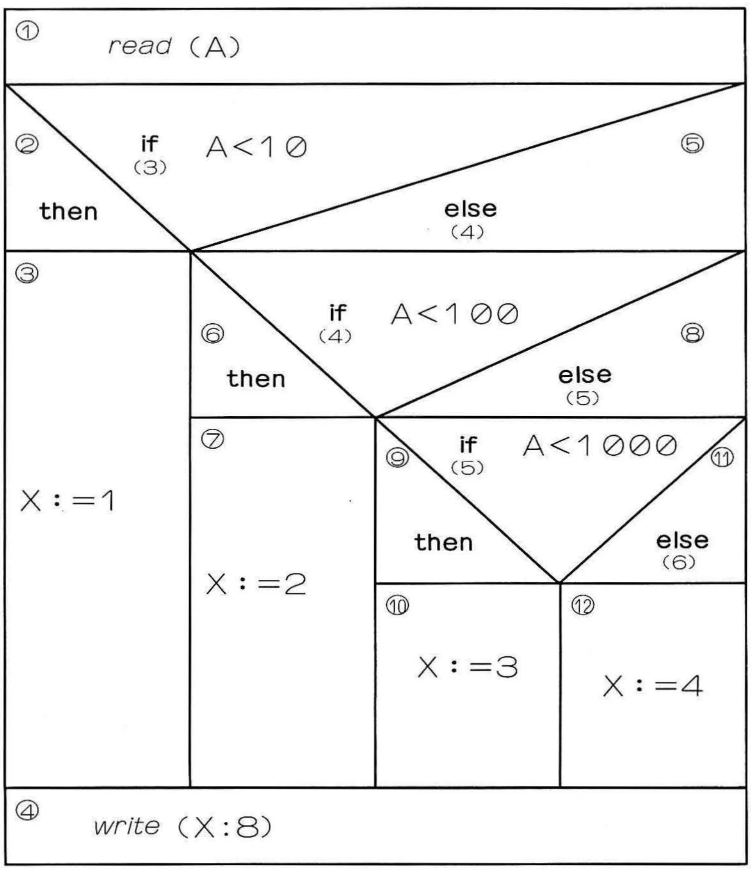



```
if A<0 then write ("END")
   else begin
          X:=sqrt (A);
          writeln (X)
        end;
```

An **if** statement including **else** is represented as shown above. Statements executed when the condition is satisfied are written on the left and statements following **else** are written on the right.
Let's try representing the exercise shown on page 59 using an NS chart.

Answer to the exercise

```
0. var A,X: integer;
1.   begin
2.     read (A);
3.     if A<10 then X:=1
4.       else if A<100 then X:=2
5.            else if A<1000 then X:=3
6.                 else X:=4;
7.     write (X:8)
8.   end.
```



When the value assigned to A is 9 or less, the execution sequence is

①→②→③→④

When the value assigned to A is 10~99, the execution sequence is

①→⑤→⑥→⑦→④

When the value assigned to A is 100~999, the execution sequence is

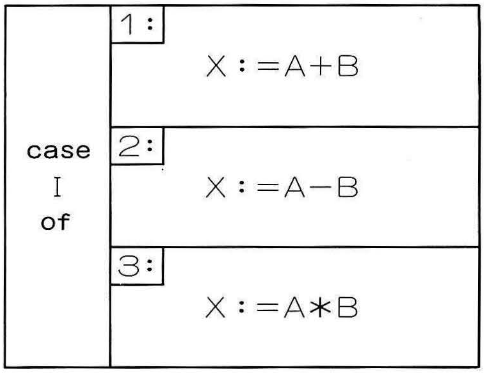
①→⑤→⑧→⑨→⑩→④

When the value assigned to A is greater than 1000, the execution sequence is

①→⑤→⑧→⑪→⑫→④

Note: The numbers in parentheses are line numbers from the program.

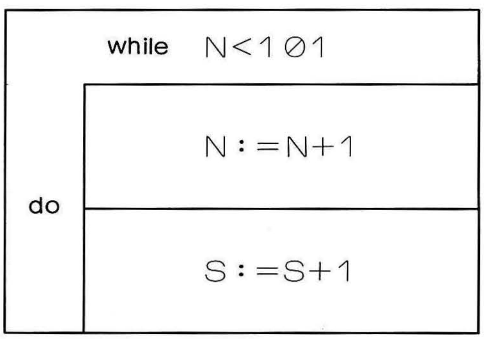
3. CASE Statement




```
case I of 1: X:=A+B;
          2: X:=A-B;
          3: X:=A*B
end;
```

The conditional expression is written on the left and the case labels and their corresponding statements are written on the right. In the above example, X:=A−B is executed when I is 2.

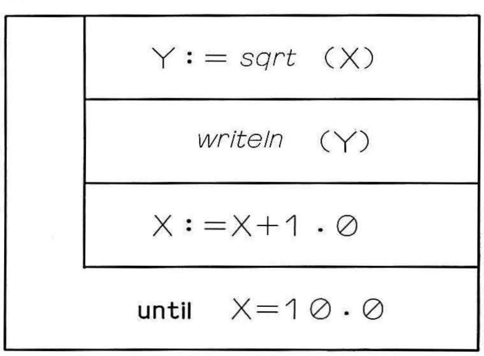
4. WHILE Statement




```
while N<1Ø1 do
  begin
    N:=N+1;
    S:=S+1
  end;
```

Since a **while** statement begins with a conditional determination, the conditional expression is written at the top. Statements repeated while the condition is satisfied are written on the right. When the condition is no longer satisfied, program execution proceeds through the left side.

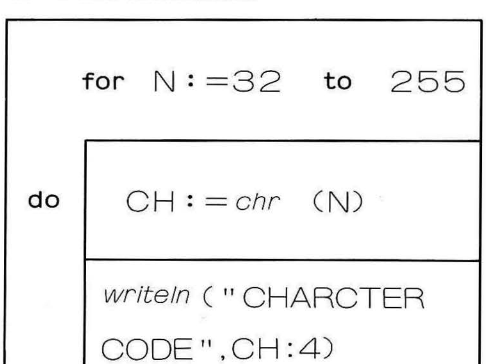
5. REPEAT Statement




```
repeat
  Y:=sqrt(X);
  writeln(Y);
  X:=X+1.Ø
until X=1Ø:Ø;
```

The form of this NS chart is the inverse of that for the **while** statement.

6. FOR Statement




```
for N:=32 to 255 do
  begin
    CH:=chr(N);
    writeln("CHARACTER CODE",CH:4)
  end;
```

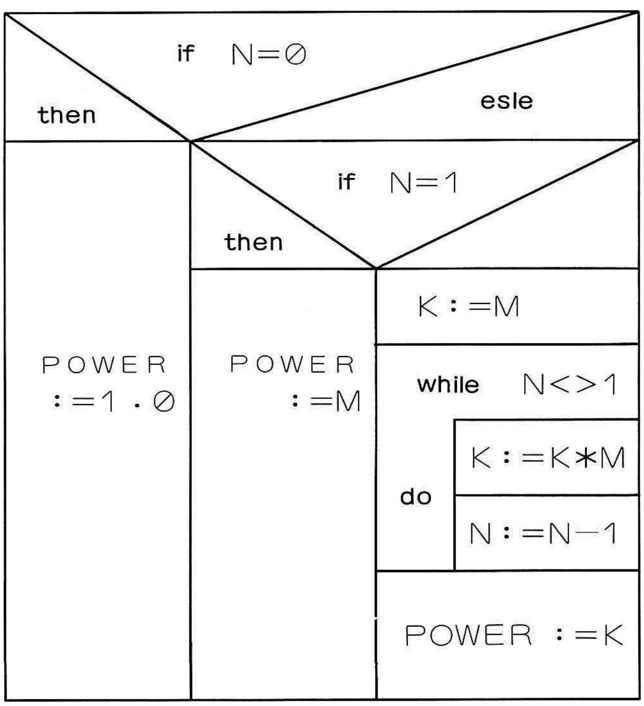
The loop condition is written at the top and statements after **do** are written on the right.

These NS charts allow the structure of a PASCAL program to be represented in a clear manner.

One NS chart is used for each procedure and function. Let's make NS charts for the following program. This program reads the value of X and Y from the keyboard, raises X to the Yth power and displays the result.

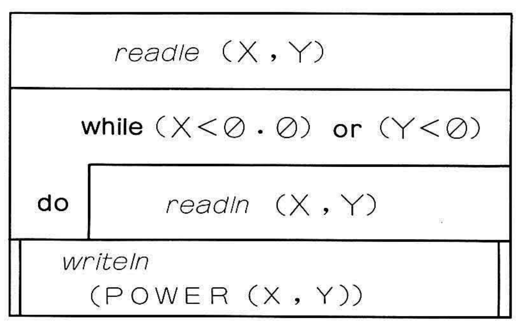
```
var X : real; Y : integer;
function POWER (M : real; N : integer) : real;
    var K : real;
        begin
          if N=0 then POWER : =1.0
              else if N=1 then POWER : =M
                     else begin
                            K : =M ;
                            while N<>1 do
                              begin K : =K*M ; N : =N-1 end ;
                            POWER : =K
                          end
        end ;
begin
    readln (X,Y) ;
    while (X<0.0) or (Y<0) do readln (X,Y) ;
    writeln (POWER (X,Y) )
end.
```

function POWER
Parameter M : *real* N : *integer*



The **if** statement in the function declaration includes another **if** statement, which includes a **while** statement. Thus, the NS chart of the function declaration is as shown at left.

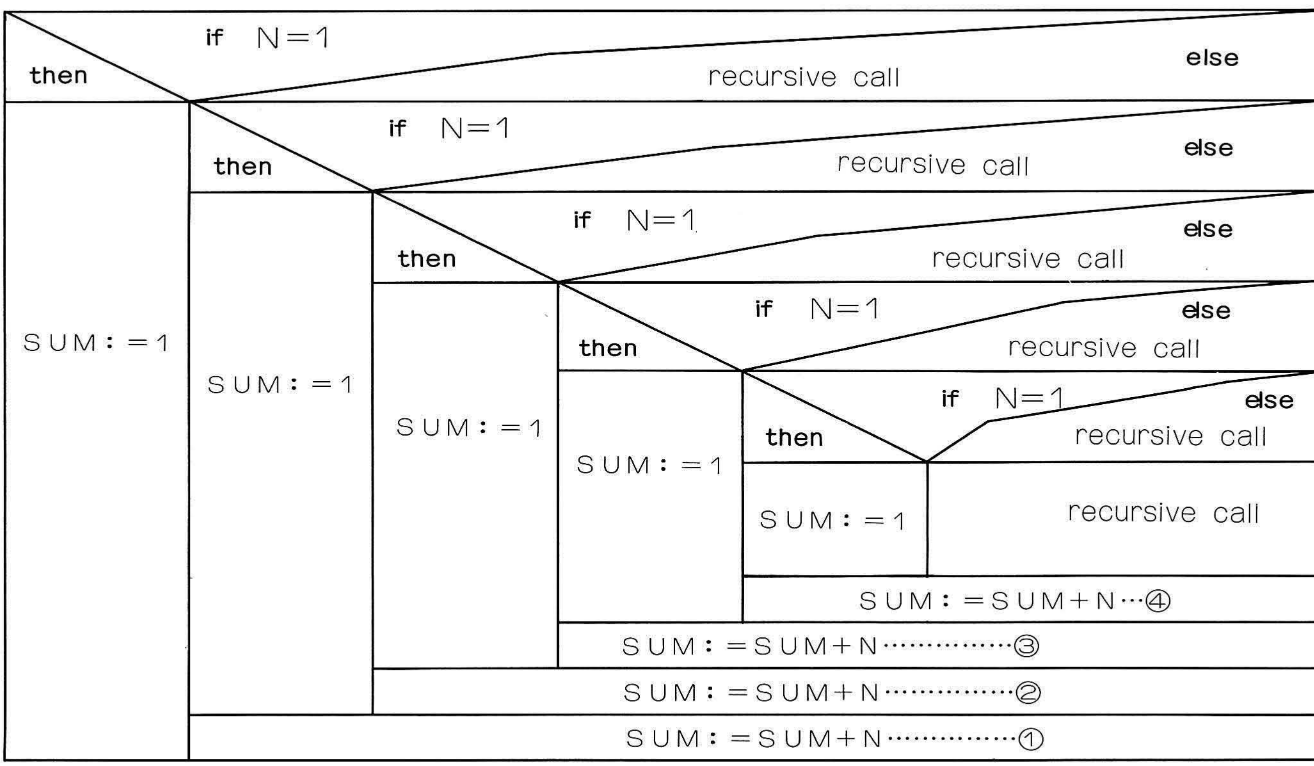
The NS chart of the main program is shown below.



Recursion can also be represented using NS charts. The following sample program gives the sum of integers 1 through N.

```
function SUM (N : integer) : integer;
  begin
    if N=1 then SUM : =1
        else SUM : =SUM (N-1) +N
  end ;
```

function WA
parameter N、result WA



Take note of the method used to specify N in SUM: =SUM+N. The value of N is saved in local variable N every time a recursive call is executed, and it is restored upon return. That is, local variable N is declared every time a recursive call is executed.

Assume that function SUM is designated when N is 5. The program executes N:=5, then performs a recursive call; N: =4 is executed and a recursive call is performed again during execution of the first recursive call; N: =3 is executed and a recursive call is performed during execution of the second recursive call; and so on.

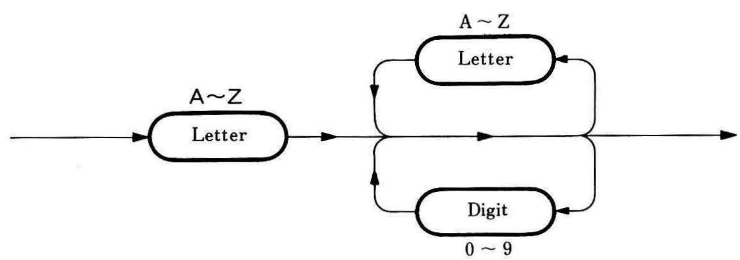
Thus, N is 5 at ①, 4 at ②, 3 at ③ and 2 at ④, and the result is 15.

Chapter 7

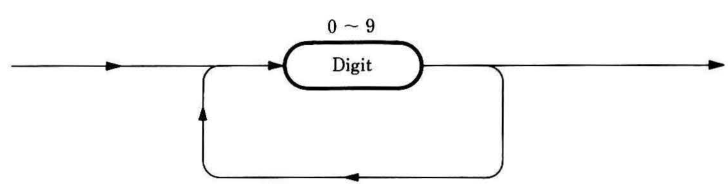
Summary

SYNTAX DIAGRAM

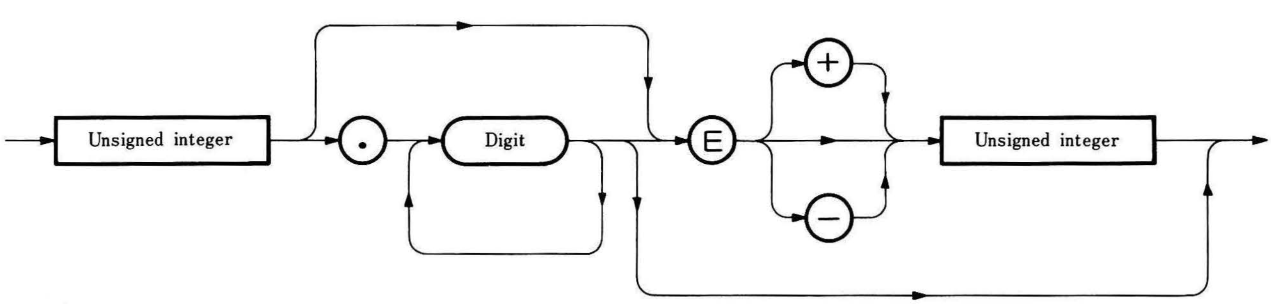
IDENTIFIER



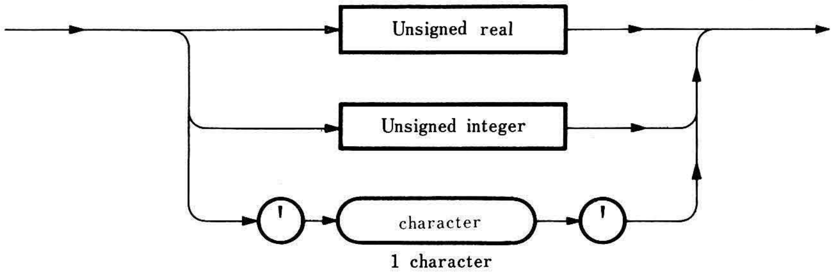
UNSIGNED INTEGER



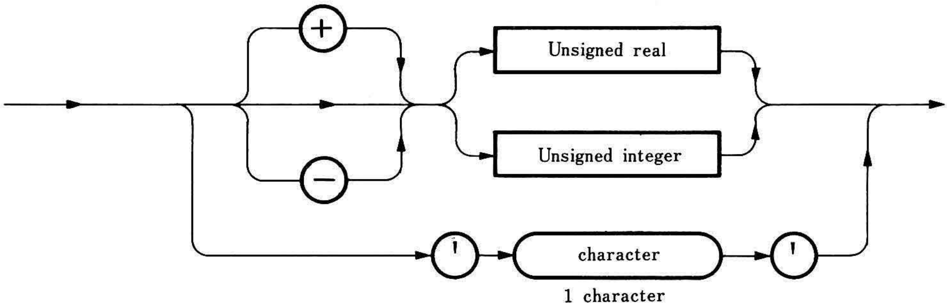
UNSIGNED REAL



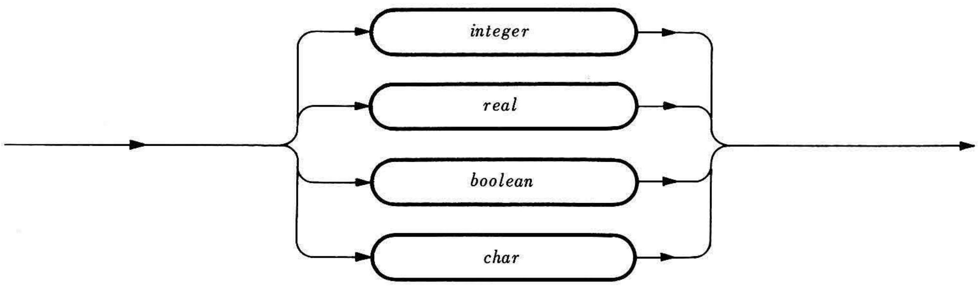
UNSIGNED CONSTANT



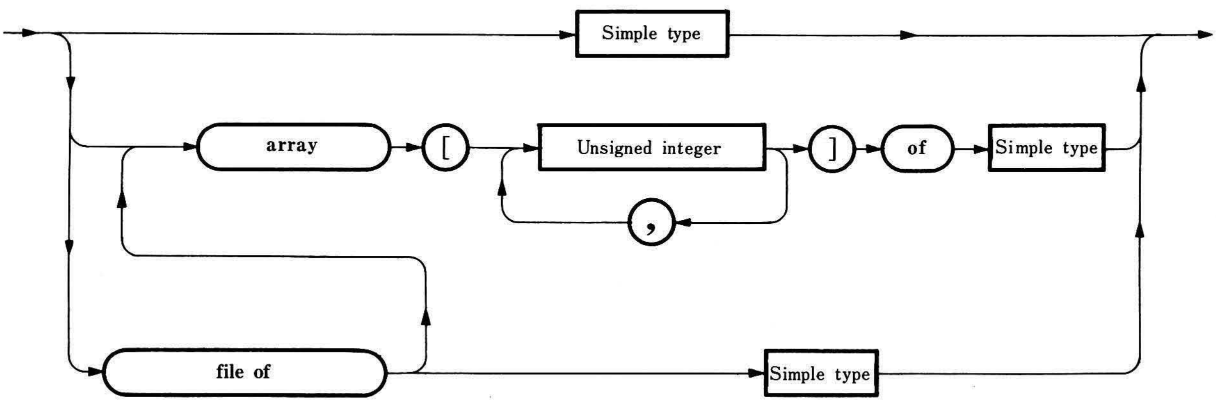
CONSTANT



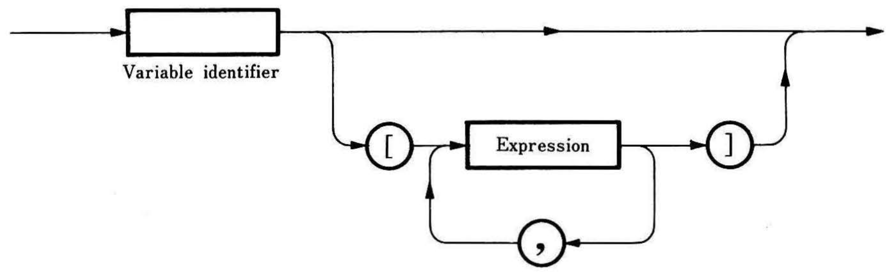
SIMPLE TYPE



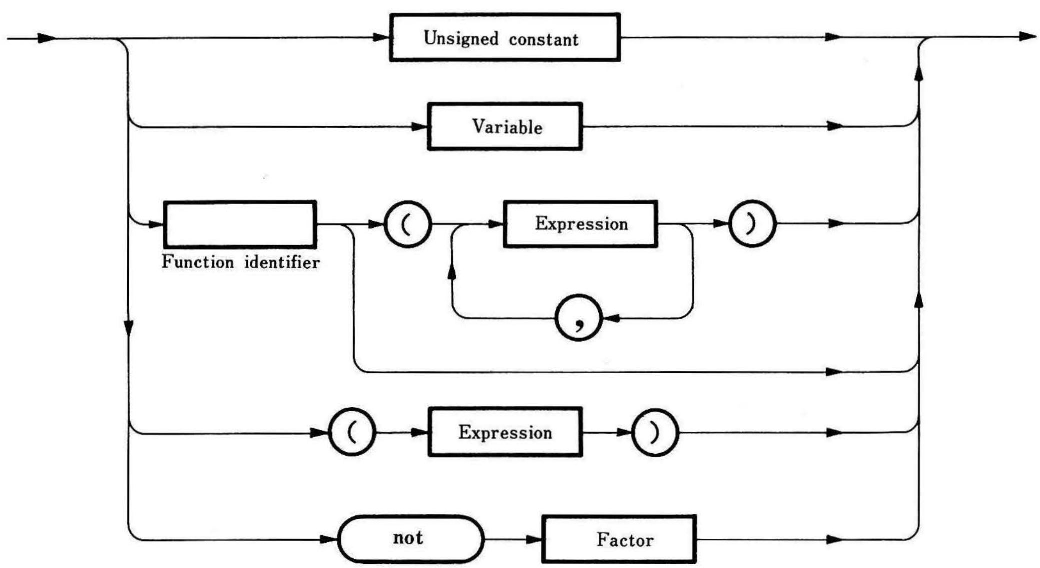
TYPE



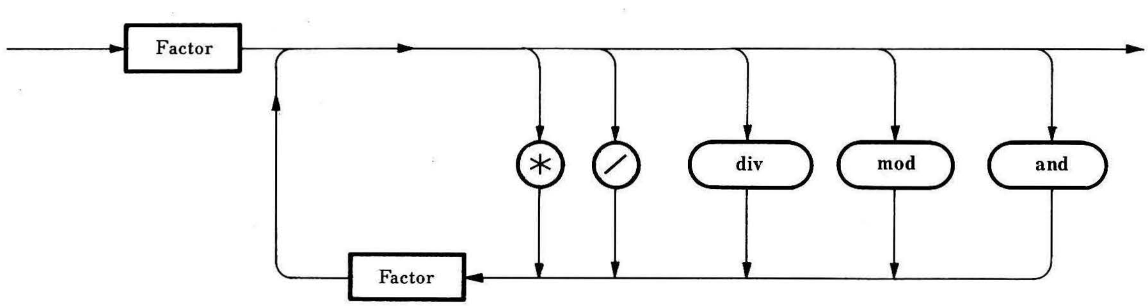
VARIABLE



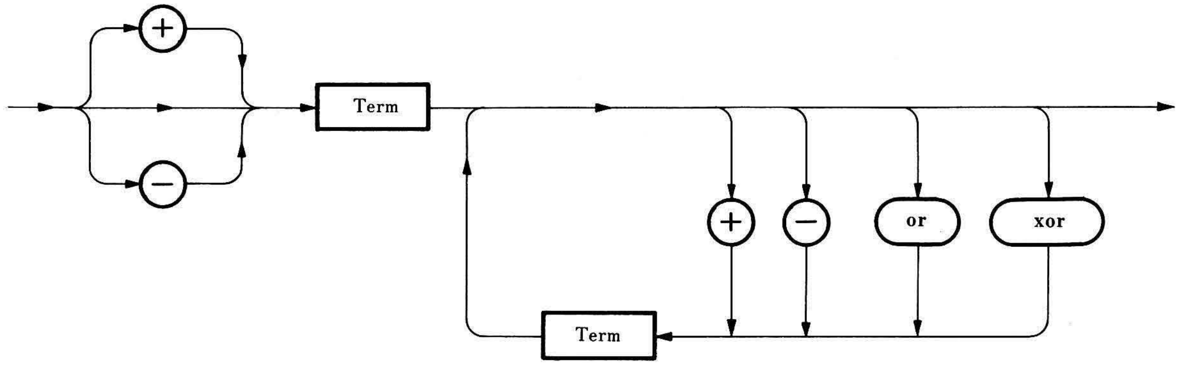
FACTOR



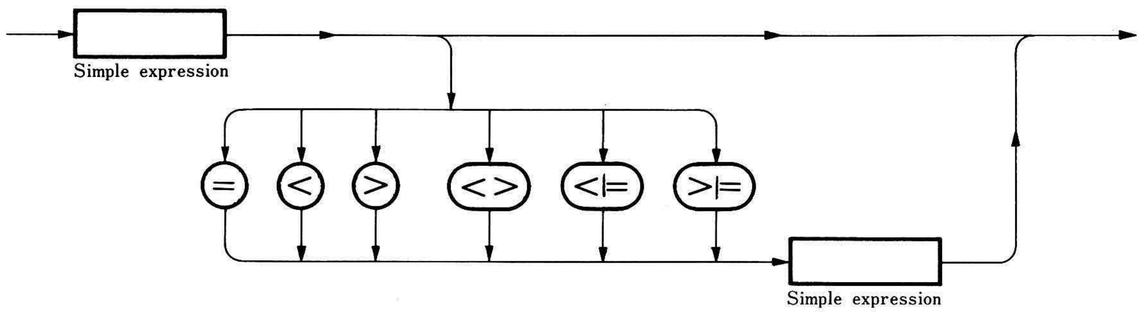
TERM



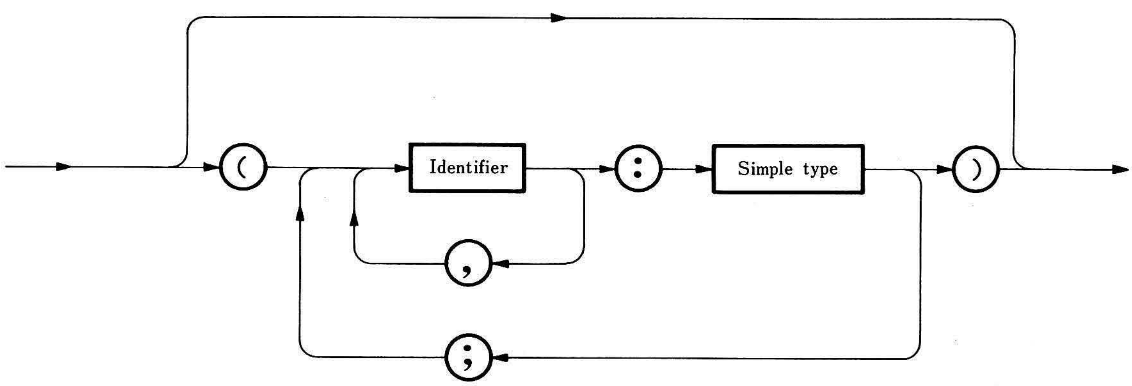
SIMPLE EXPRESSION



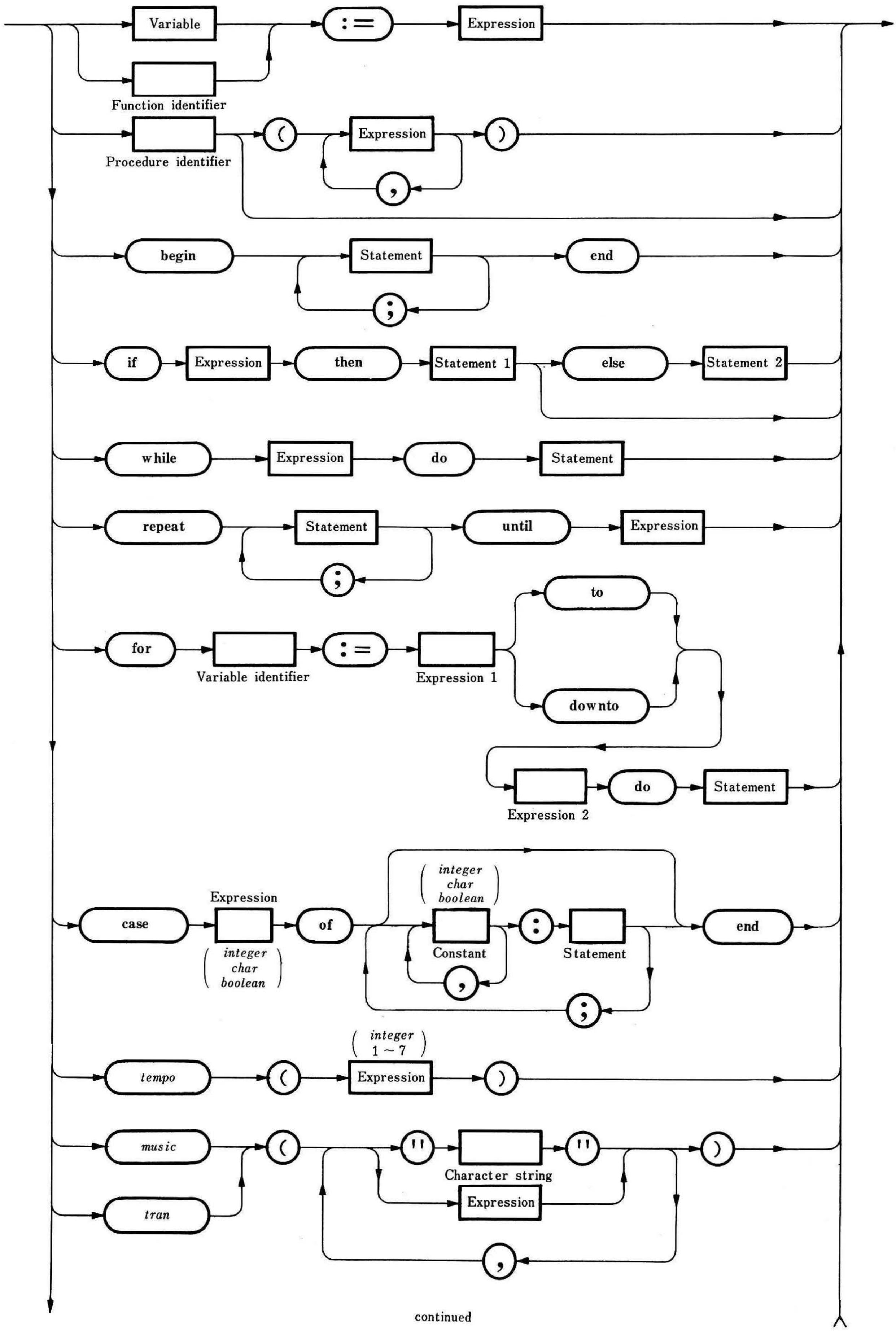
EXPRESSION



PARAMETER LIST

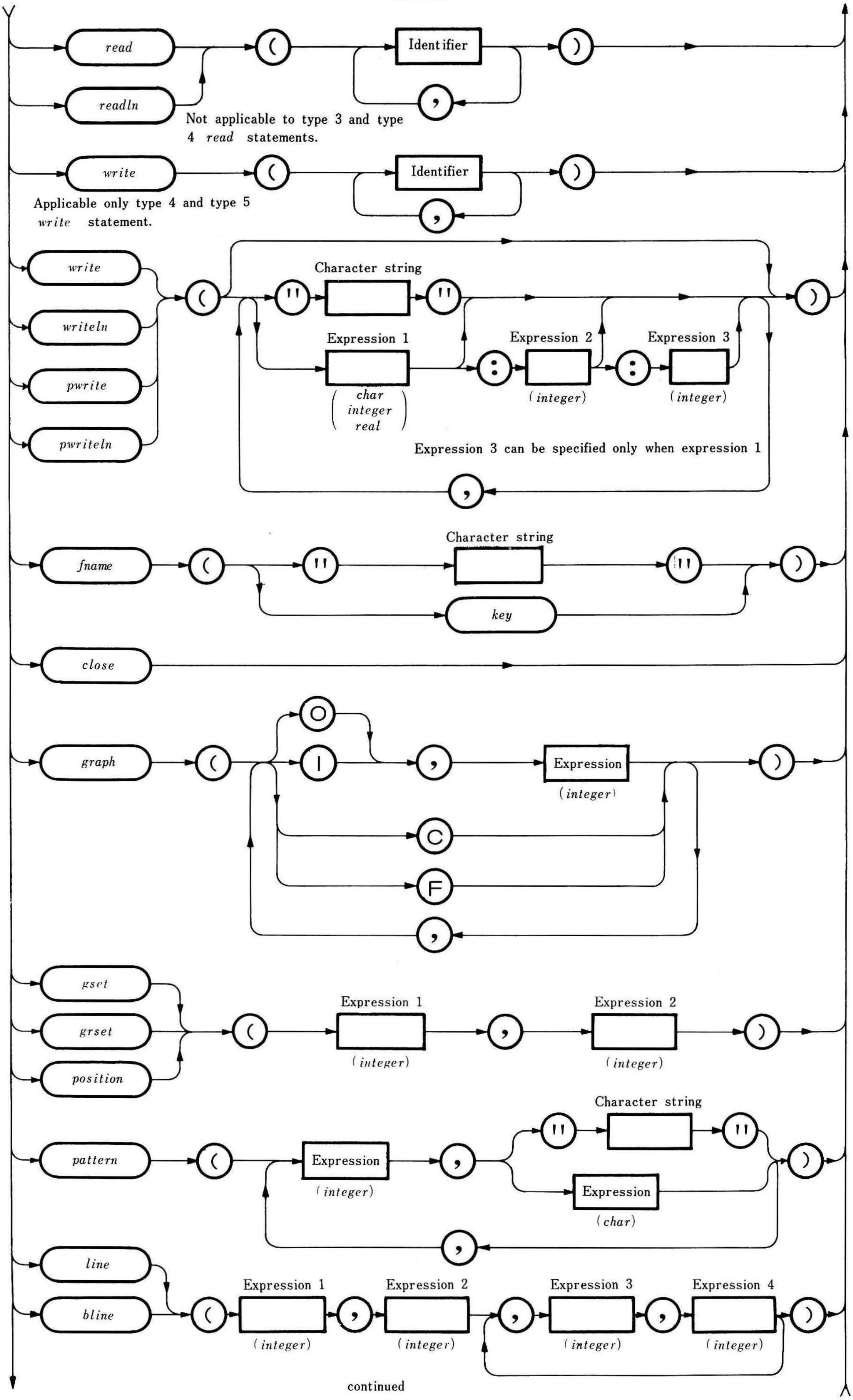


STATEMENT

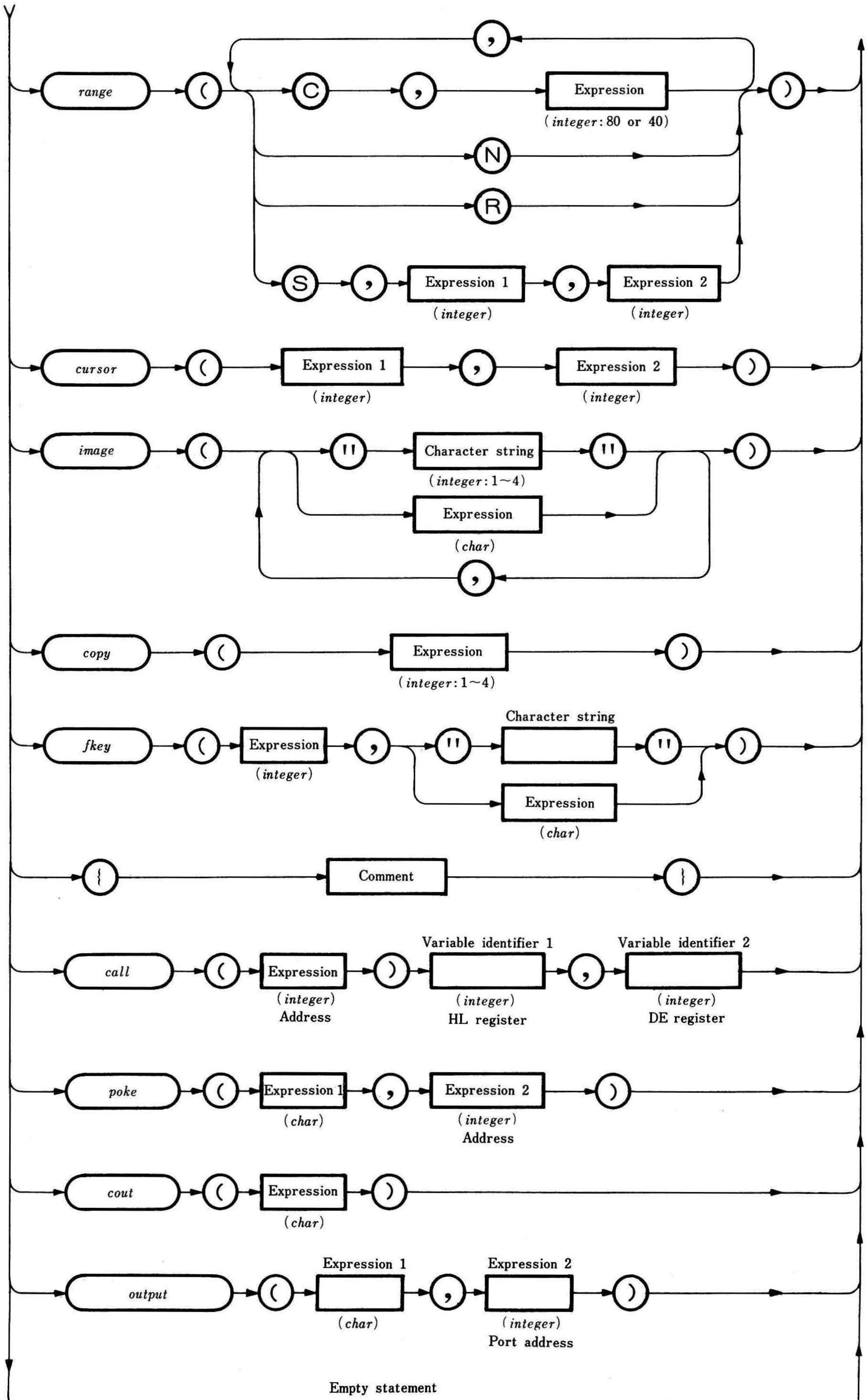


continued

continued

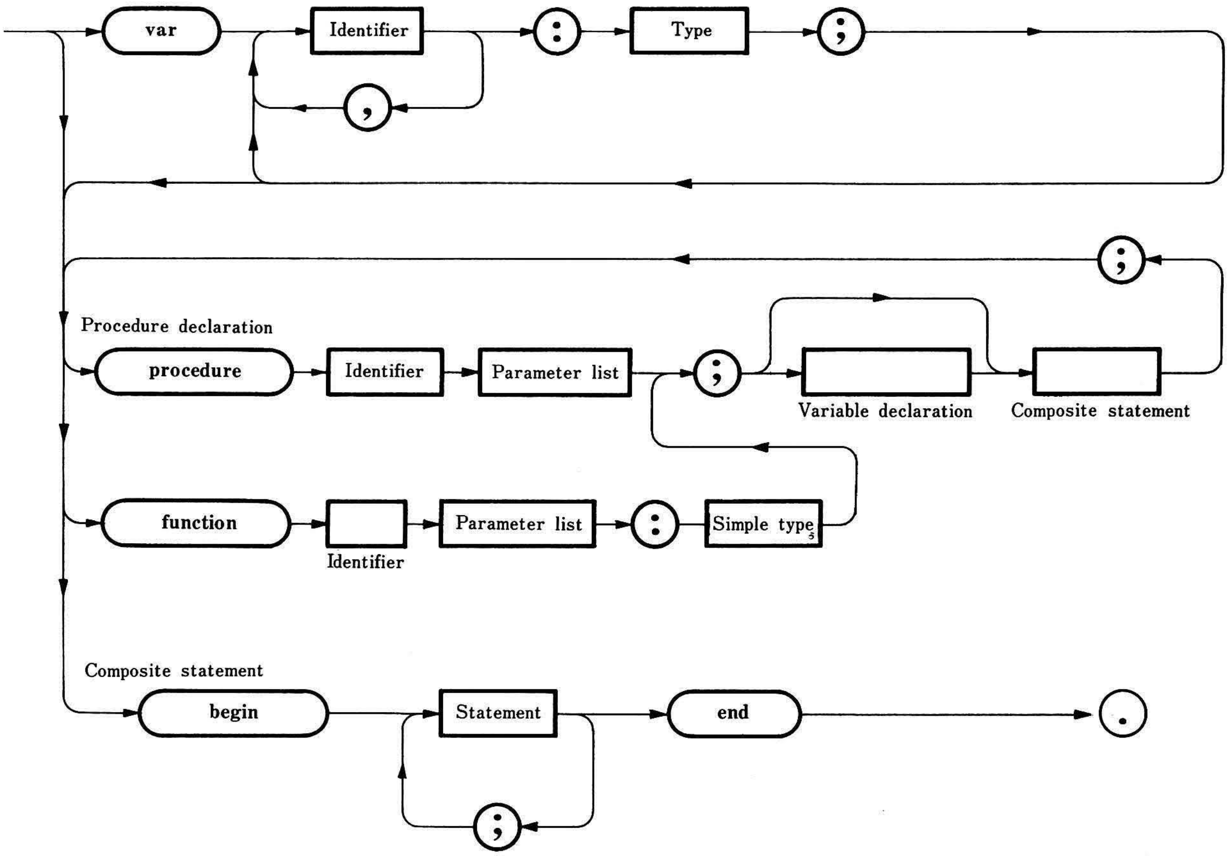


continued



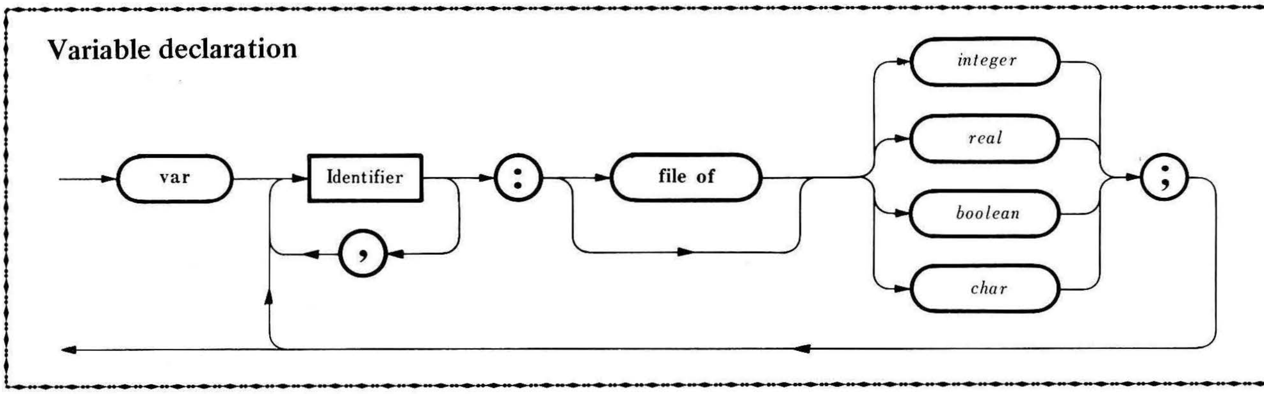
range
(
C
,
Expression
(integer: 80 or 40)
N
R
S
,
Expression 1
(integer)
,
Expression 2
(integer)
)
cursor
(
Expression 1
(integer)
,
Expression 2
(integer)
)
image
(
''
Character string
(integer: 1~4)
''
Expression
(char)
,
)
copy
(
Expression
(integer: 1~4)
)
fkey
(
Expression
(integer)
,
''
Character string
''
Expression
(char)
)
{
Comment
}
call
(
Expression
(integer)
Address
)
Variable identifier 1
(integer)
HL register
,
Variable identifier 2
(integer)
DE register
poke
(
Expression 1
(char)
,
Expression 2
(integer)
Address
)
cout
(
Expression
(char)
)
output
(
Expression 1
(char)
,
Expression 2
(integer)
Port address
)
Empty statement

BLOCK


var
Identifier
:
Type
;
,
;
Procedure declaration
procedure
Identifier
Parameter list
;
Variable declaration
Composite statement
function
Parameter list
:
Simple type
Identifier
Composite statement
begin
Statement
end
.
;

Summary of syntax

1 . Variable declaration



There are two types of variables: global variables and local variables. The former is declared in the variable declaration at the beginning of a program and the latter is declared in a procedure or function declaration. Global variables are significant throughout the program and local variables are significant only within the procedures and functions in which they are declared.

Example 1 :

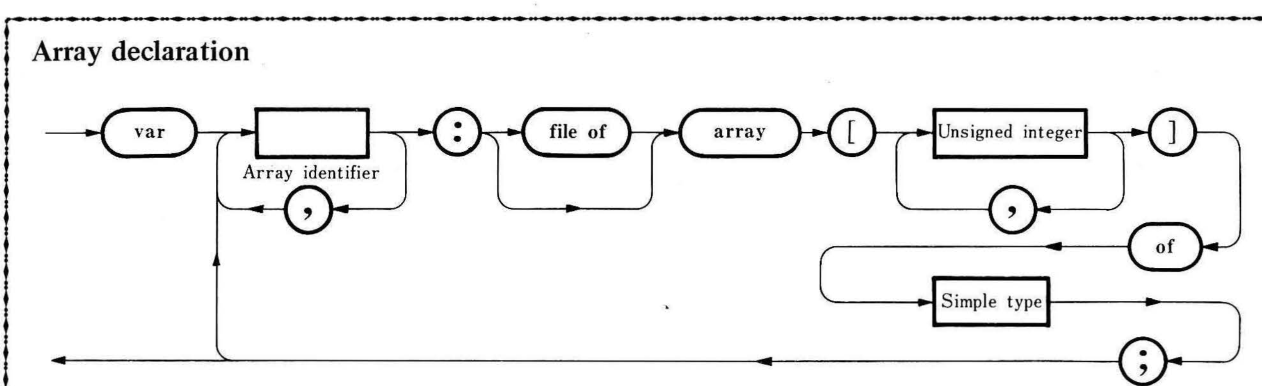
var A, B, SHARP : *integer* ; . . . . *integer* variable declaration
C, D, DATA : *real* ; . . . . *real* variable declaration
E, JUDGMENT : *boolean* ; . . . . *boolean* variable declaration
CH, MESSAGE : *char* ; . . . . *char* variable declaration

Example 2 : File declaration

var X, Y : **file of** *real* ; . . . . *file* declaration

Note: *integer* variables range from 0 to ±32767 in decimal notation; only one character can be assigned to a *char* variable; *boolean* variables take only the values *true* and *false*; and *real* variables range from ±0.27105055E–19 to ±0.92233720E+19.
No variables declared as *file* can be used as parameters.

2 . Array declaration



Arrays are declared in the variable declaration section. The size of arrays differs according to data type. The number of dimensions of an array is not limited. Arrays can be declared in a local variable declaration section.

Example 1 :

var A : **array** [10] **of** *integer* ; .... One-dimensional array declaration
DATA : **array** [100, 10] **of** *real* ; .... Two-dimensional array declaration
SHARP : **array** [10, 5, 5] **of** *real* ; .... Three-dimensional array declaration
MZ : **array** [10, 5, 5, ..., n] **of** *char* ; .... N-dimensional array declaration

Example 2 : *file* declaration

var X : **file of array** [50] **of** *real* ;
Y : **file of array** [100, 5] **of** *char* ;

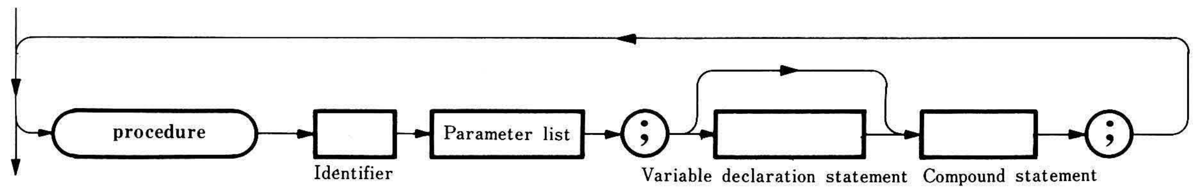
Example 3 : Simultaneous declaration of arrays and variables

0. **var** DATA : **array** [100, 10] **of** *real* ; .... Array declaration
1. A : **file of array** [100] **of** *integer* ; .... Array declaration
2. B : *boolean* ; .... Variable declaration
3. CH, PRINT : *char* ; .... Variable declaration

Note: The indexes of arrays must be positive integers.

3 . Procedure declaration

Procedure declaration



Procedures must be declared in the procedure declaration section. Local variables are declared in each procedure declaration. Their name may be the same as those used for global variables.

Example 1 : when no parameters are used.

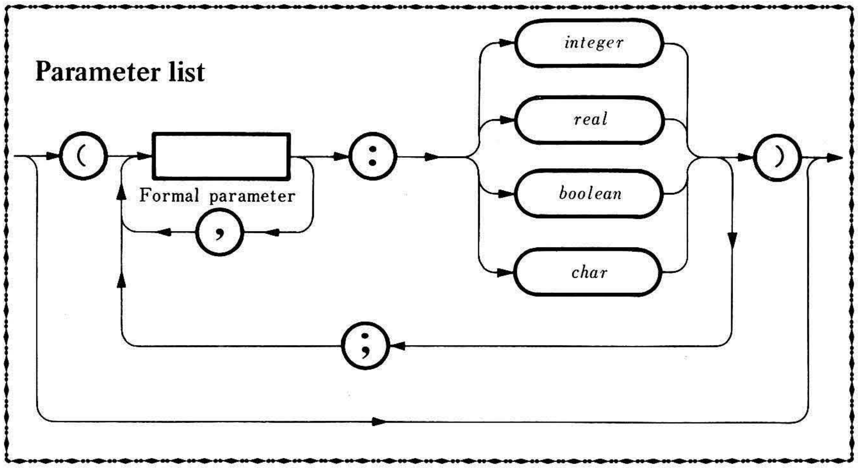
procedure DATAOUT ; .... Declares DATAOUT as a procedure identifier.
var N : *integer* ; .... Declares local variable N.
begin
for N : = 0 **to** 9 **do** *write* (DATA [N]) .... Displays array data.
end ;

Example 2 : when parameters are used.

procedure MULTI
(X, Y : *real*);
begin
Z : =X ✱ Y
end ;

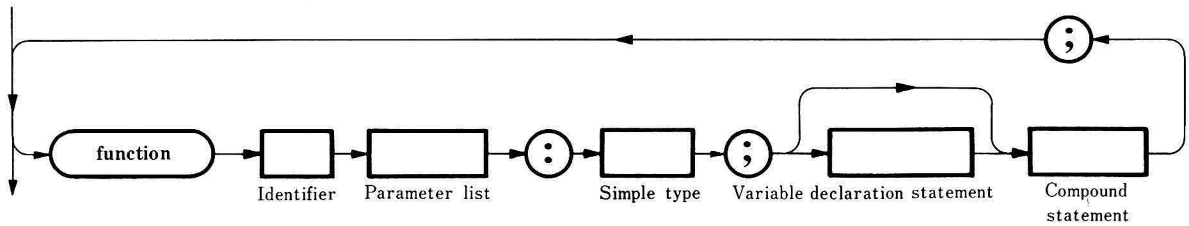
No file identifiers can be specified as parameters.

Parameter list



4 . Function declaration

Function declaration



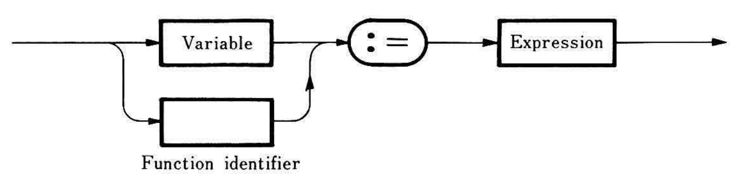
Functions must be declared in the function declaration section.

Example: **function** AREA (A, H : *real*) : *real* ;
begin
AREA : = (A∗H) / 2.0
end ;

Local variables are declared in each function declaration.

5 . Assignment statement

Assignment statement



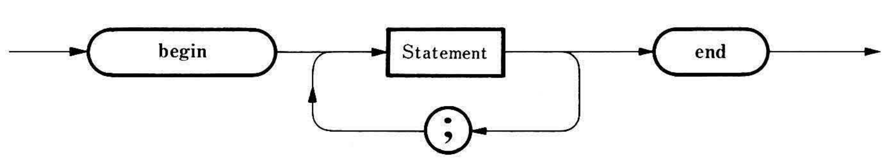
The assignment statement assigns the result of the right member to the variable or function.

Example: A : =5 . . . . "A" must be an *integer* variable.
B : =5.0 . . . . "B" must be a *real* variable.
Z : =(X>Y) . . . . "Z" must be a *boolean* variable.
C : ='A' . . . . "C" must be a *char* variable.

Note: 1. STR1 : ="ABC" Incorrect because a character string cannot be assigned to a *char* variable.
2. STR2 : = 'ABC' Incorrect because only one character can be placed between single quotation marks.

6 . Compound statement

Compound statement



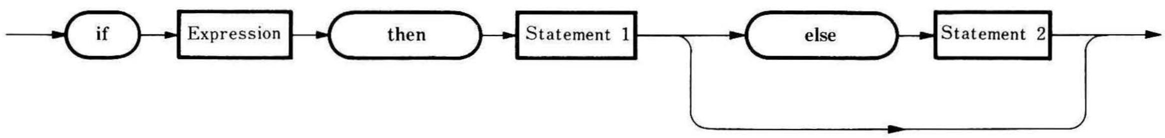
A compound statement consists of many statements. It starts with **begin** and ends with **end**.

Example: **begin** M : =2∗M ; SUM : =SUM+M **end** ;

Although only one statement is allowed after **do**. **then** and **else**, a number of statements can be combined and written as one compound statement.

7 . I F statement

if statement



Example 1 : **if** A>B **then** A : =A−1 ;

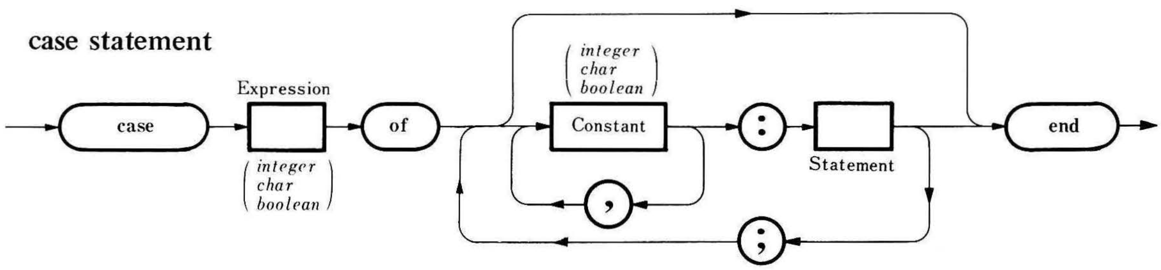
If A>B gives *true*, A : =A−1 is executed, otherwise, execution continues with the next statement.

Example 2 : **if** A>B **then** A : =A−1;
else B : =B−1

If A>B is *true*, A : = A − 1 is executed; if *false*, B : = B − 1 is executed. Only one statement can be specified after **then** or **else**. Use a compound statement if two or more statements are required.

8 . CASE statement

case statement



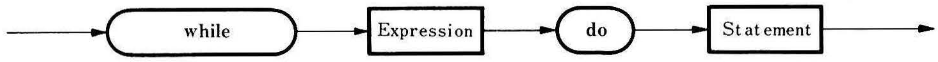
Executes the statement with the case label indicated by the expression. If the case label does not exist, the next statement is executed. The value of the label must be within the range −32767 ~ +32767 when the expression is of the *integer* type, one listed in the ASCII code table when it is of the *char* type and *true* or *false* when it is of the *boolean* type.

Example: **case** I **of** 1 : X : =A+B ;
(I = Expression) 2 : X : =A−B ;
3 : X : =A∗B

end ;

9. WHILE statement

while statement

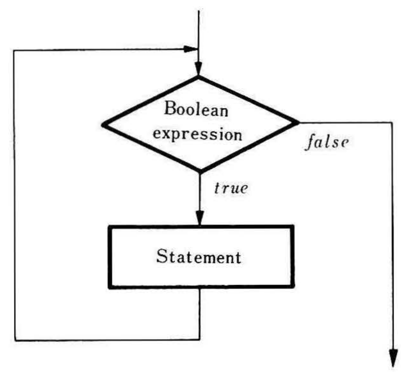


The statement after **do** is repeated if the expression between **while** and **do** is *true*, otherwise, the next statement is executed. The expression gives *false* from the beginning, the loop is not performed.

Only one statement can be specified after **do**; use a compound statement to execute two or more statements.

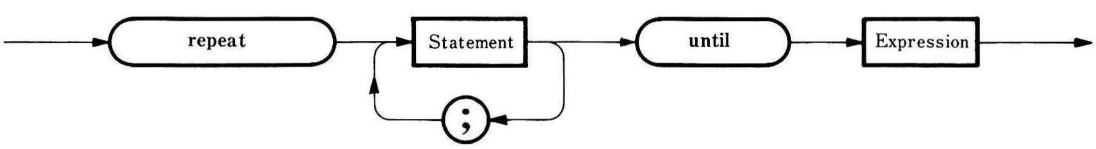
Example :

```
while  X<>0  do
    read (X) ;
```



10. REPEAT statement

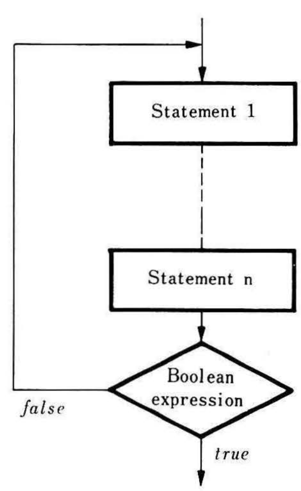
repeat statement



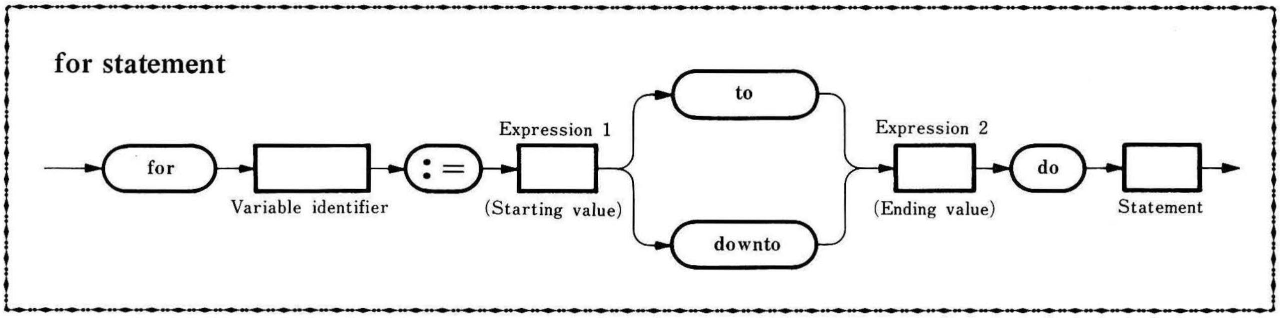
The statement between **repeat** and **until** is executed first, then the result of the expression after **until** is checked. If the result is *false* the statement is repeated; otherwise, the next statement is executed. The statement is executed at least once even if the result of the expression is *true* from the start.

Many statements can be specified between **repeat** and **until**; it is not necessary to use compound statements.

```
repeat
  read (A) ;
  X : =X+A
until  A=0 ;
```



11. FOR statement



Example 1 :

for N : = 1 **to** 10 **do** *write* ("A") ;

Assigns 1 to N as the starting value, repeats the statement following **do** with N incremented by 1 for each repetition until N becomes 10.

In this case, 10 "A's" are displayed on the screen.

Example 2 :

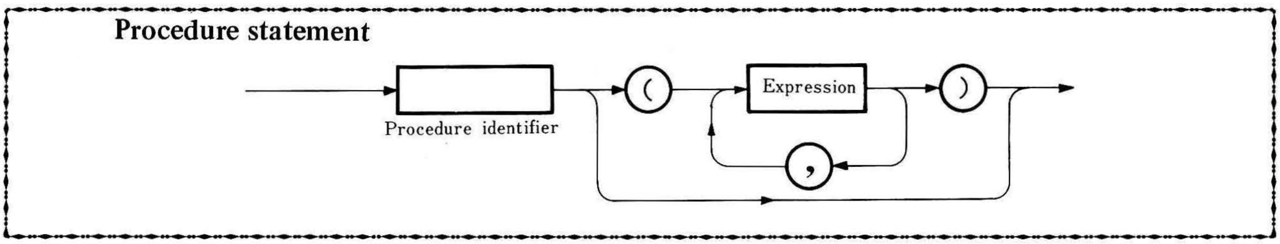
for N : = 15 **downto** 1 **do** *write* ("A") ;

The starting value of N is 15. The statement following **do** is repeated with N decremented by 1 for each repetition until N becomes 1.

In this case, 15 "A's" are displayed on the screen.

Only one statement can be specified after **do**. Use a compound statement to execute two or more statements.

12. Procedure statement



Calls a declared procedure. There are two types of procedure statement: one accompanies parameters and the other does not.

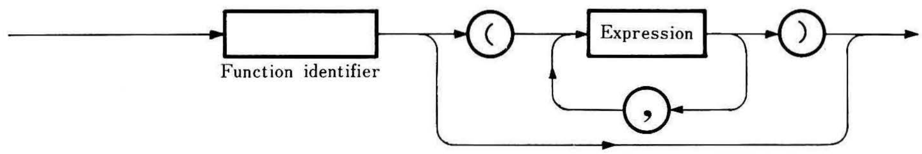
Example:

DATAIN . . . . Calls the procedure DATAIN.
SELECT (M) . . . . Calls the procedure SELECT with parameter M assigned to the formal parameter.
CURSOR (X, Y) . . . . Calls the procedure CURSOR with parameters X and Y assigned to formal parameters.

Note: The type of each actual parameter must be the same as that of the corresponding formal parameter.
No file identifier can be specified as a parameter.

13. Function designation

Function designation



Program control is returned to the statement which calls the function with the result assigned to the function identifier. Otherwise this function is similar to the procedure statement.

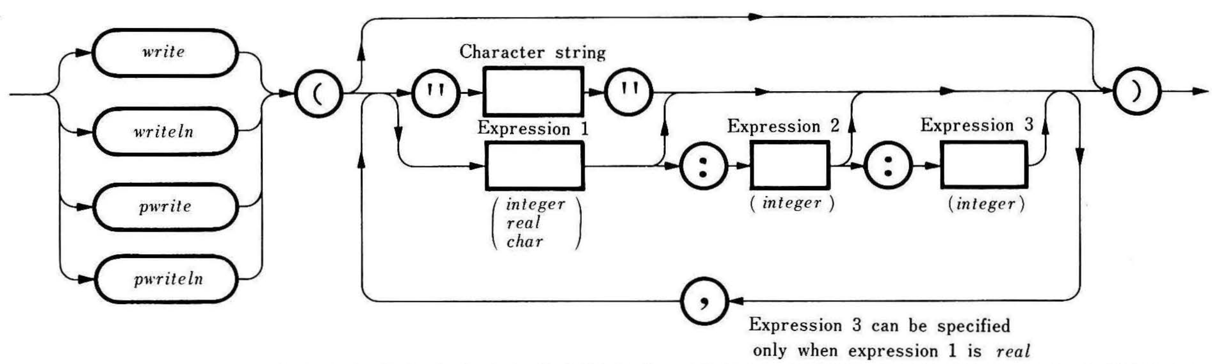
Example :

FACTORIAL (N) . . . . . . The function FACTORIAL is called with N assigned to the formal parameter. The result is assigned to FACTORIAL. N must not be declared as *file*.

14. WRITE statement

Types 1, 2 and 3

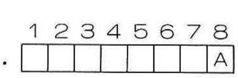
write statement



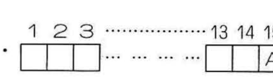
This statement displays data or a messge on the CRT screen, outputs it to the printer or writes data on cassette tape. The codes used are ASCII codes.

write	Displays data on the CRT screen. Performs no carriage return.
writeln	Displays data on the CRT screen. Performs a carriage return after display.
pwrite	Prints data on the printer. Performs no carriage return.
pwriteln	Prints data on the printer. Performs a carriage return after printing.

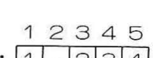
write ('A' : 8)
Displays the character "A" at the 8th position from the current cursor position. . . . .



wirte ('A')
The default value of expression 2 is 15. . . . .



When X is a *real* variable and the data is 1.23456,
write (X : 5 : 3) displays . . . .

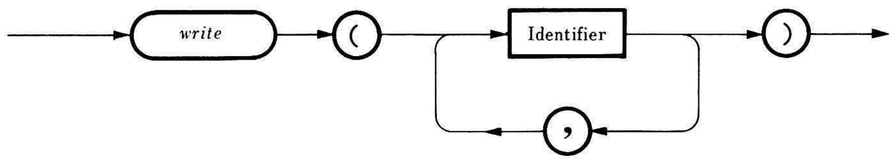


Expression 3 specifies the number of decimal places.

For *write* (X, Y, Z), the file declaration is only checked for X. When X is not declared as *file*, Y and Z are assumed to be other than *file* also.

Types 4 and 5

write statement (type 4 and 5)

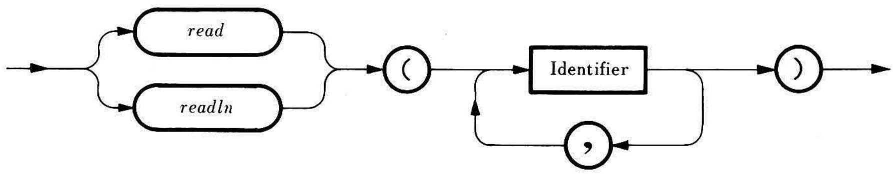


write statements of types 4 and 5 store variable and array data in the cassette tape file. No expression can be specified within parentheses. For *write* (X, Y, Z), an error results when X is declared as *file* but Y and Z are not. File declaration is checked only for X.

15. READ statement

Types 1 and 2

read statement (type 1 and 2)



read . . . . Reads data from the keyboard. Performs no carriage return after reading.
readln . . . . Reads data from the keyboard. Performs a carriage return after reading.

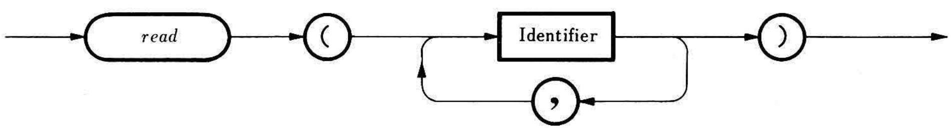
A *read* statement specified for *char* variables cannot read a character string. 13 ($0D) is assigned to X when only the CR key is pressed for *read* (X). For *read* (X, Y, Z), file declaration is checked only for X. Therefore, when X is declared as *file*, Y and Z are assumed to be declared as *file* even if they are not.

readln (X, Y, Z) performs a carriage return after the last data has been read. No expression can be specified in parentheses. 2 CR may be keyed when *read* (X) is executed and X is a *real* variable. The data is converted into 2.0 internally.

For *read* (X, Y, Z), the CR key is not required to be pressed after the entry for each variable; press 5 , 6 , and CR .

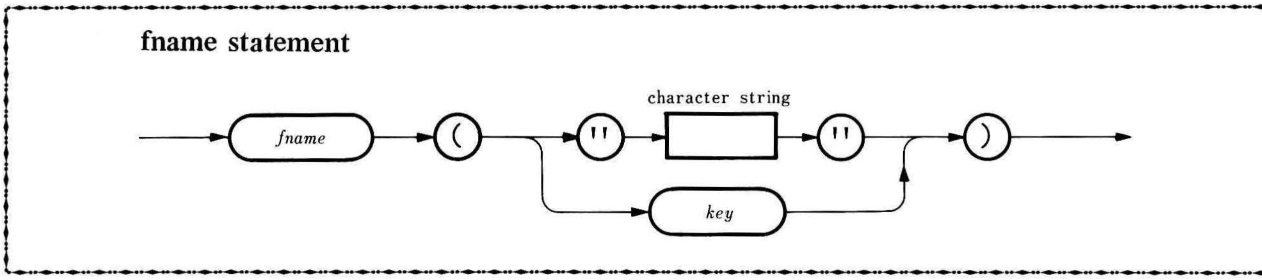
Types 3 and 4

read statement (type 3 and 4)



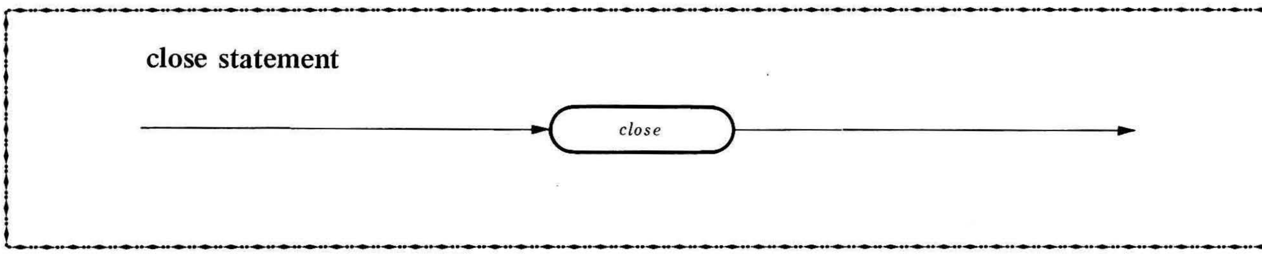
These statements read data from the cassette tape into variables and arrays. No expression can be specified in parentheses. For *read* (X, Y, Z), an error results when X is declared as *file* and Y and Z are not.

16. FNAME statement



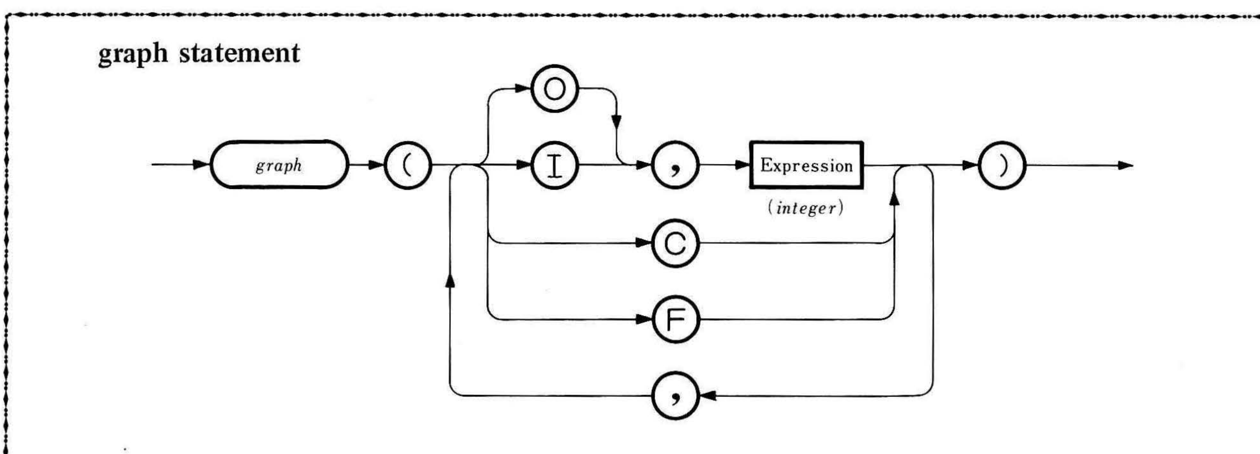
This statement opens a cassette file to allow a sequential data file to be written on or to be read from cassette tape. "⟨Character string⟩" specifies the name of the sequential data file. When the *key* function is specified instead of "⟨character string⟩", the system allows entry of file name from the keyboard.

17. CLOSE statement



This statement closes a cassette tape file opened by the *fname* statement. Closing a cassette file allows *fname* statement to be declared for other data files.

18. GRAPH statement



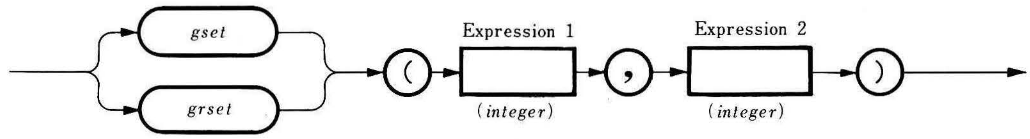
This statement sets the graphic input or output mode and clears or fills the graphic memory area.

Example: *graph* (O, 0, I, 2, C, I, 1, F, O, 3)

Clears graphic data from the display, puts graphic area 2 in the input mode, clears graphic area 2, puts graphic area 1 in the input mode, fills graphic area 1 and puts graphic areas 1 and 2 in the output mode.

19. GSET/GRSET statement

gset / grset statements



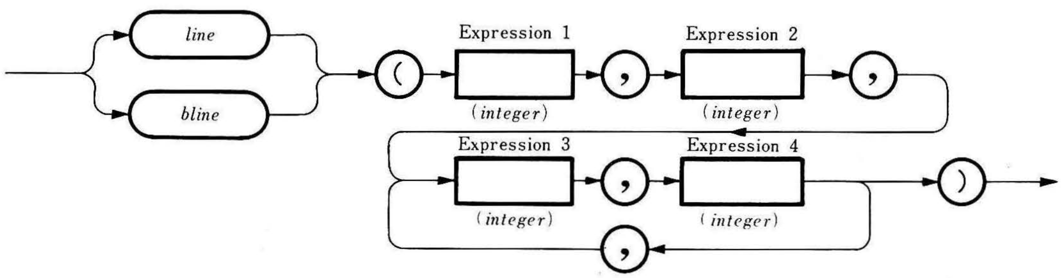
These statements set or reset a dot in any position in a graphic area operating in the input mode. The dot position is specified with X- and Y-coordinates. The X-coordinate of the graphic area can range from 0 to 319 – from left to right – and the Y-coordinate from 0 to 199 –from top to bottom.

Example: *gset* (160, 100)

Displays (Sets) a dot in the center of the screen.

20. LINE/BLINE statement

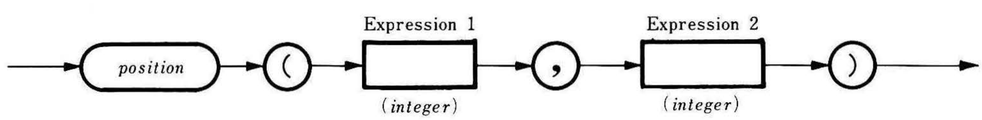
line / bline statements



These statements draw a line or a black line in the graphic area that is in the input mode, by setting dots from the first set of coordinates to the second set of coordinates. When the operand specifies three or more sets of coordinates, the system draws corresponding segments one after another.

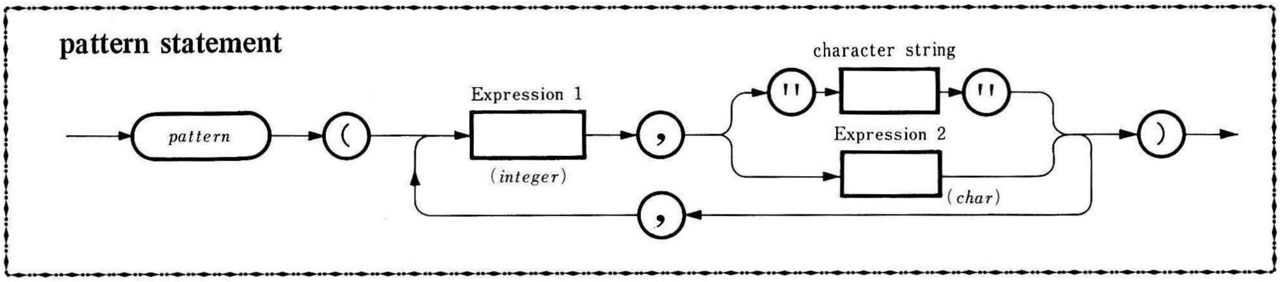
21. POSITION statement

position statement



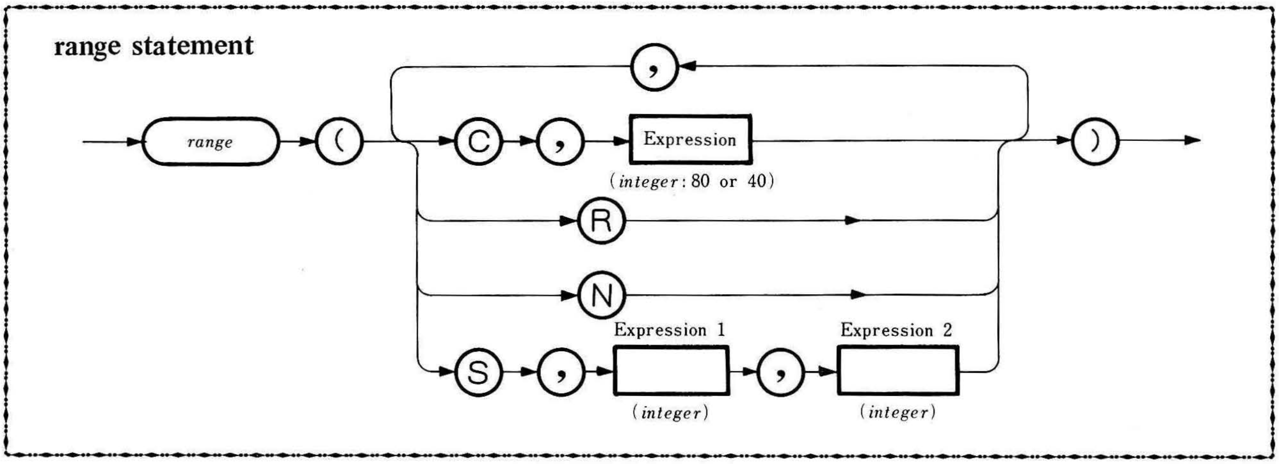
This statement sets the location of the position pointer in the graphic area. The *pattern* statement is executed starting at position coordinates indicated by the position pointer.

22. PATTERN statement



This statement draws the dot pattern specified by operands in a graphic area which is the input mode. Each dot pattern unit consists of 8 dots arranged horizontally and corresponds to 8 bits representing a character. Elements are stacked in the number of layers specified by the value of the operand (Expression 1) and the direction in which layers are stacked is specified by the sign of the value.

23. RANGE statement

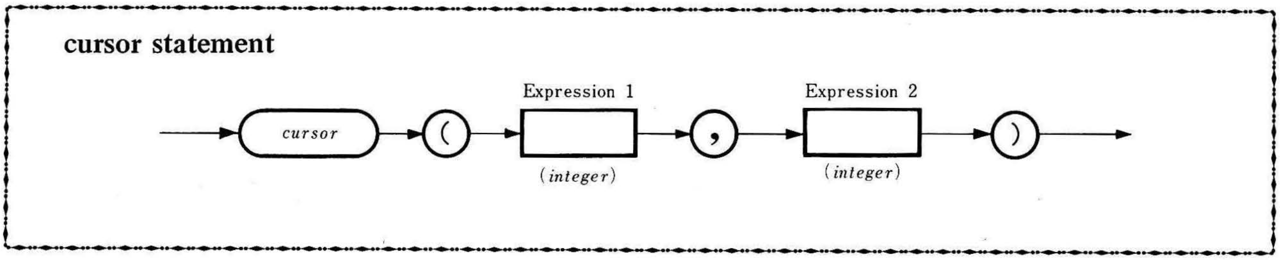


This statement changes the character display mode between 80 and 40 characters/line, between reverse mode and normal mode, or fixes the scrolling area of the display.

Example: *range* (C, 80, S, 10, 15)

Sets the display in the 80 characters/line mode and scrolling area to lines 10 through 15.

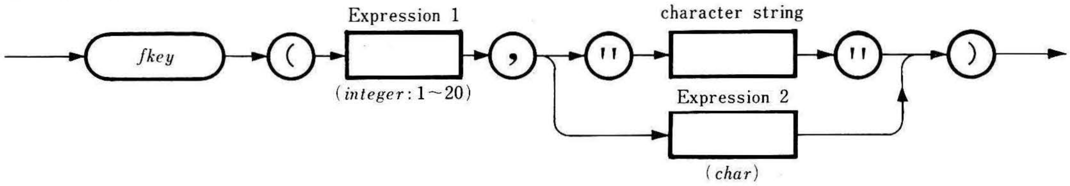
24. CURSOR statement



This statement positions the cursor on the display. – Messages issued by a *write* or a *read* statement appear beginning at the cursor position.

25. FKEY statement

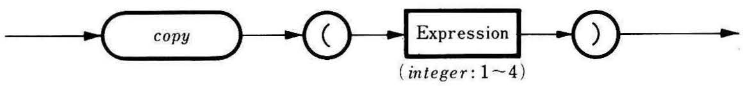
fkey statement



This statement defines function for any of the ten function keys. A number from 1 through 20 is defined to expression 1 (function number). A number from 1 through 10 is used to specify each of the function keys in normal state, and a number from 11 through 20 is used to specify each of these keys in shifted state.

26. COPY statement

copy statement



This statement causes the printer to copy an entire frame of data displayed on the computer screen.

Example: *copy* (1)

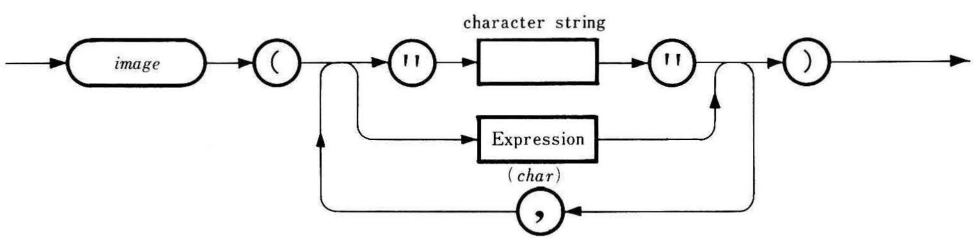
Causes the printer to copy the character display.

copy (4)

Causes the printer to copy the dot pattern set in both graphic area 1 and graphic area 2.

27. IMAGE statement

image statement

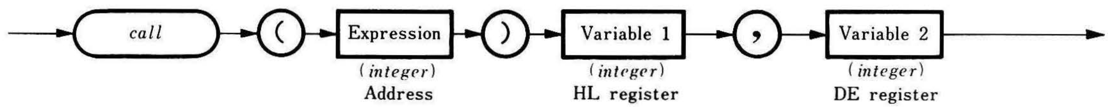


This statement causes the printer to draw a desired dot pattern according to the operating mode.

28 . CALL statement

This statement calls a user coded subroutine.

call statement

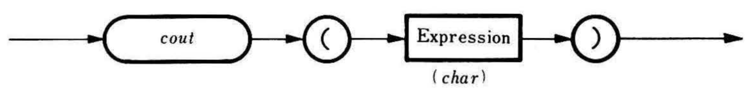


Example: *call* (X+Y) B, C

The value of B is loaded into the HL register and the value of C is loaded into the DE register, control is transferred to the address indicated by X+Y. The expression and variables may be declared as *file*.

29 . COUT statement

cout statement

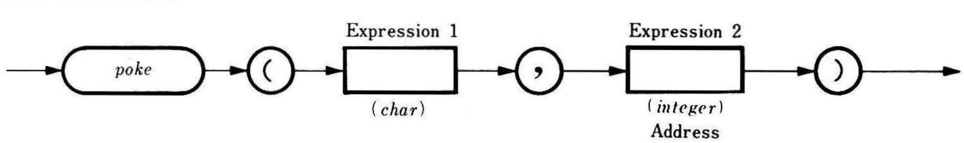


This statement displays a character at the current cursor position. The expression is of the *char* type and the codes are CIN/COUT codes. It is recommended that this statement be used in conjunction with the *cin* function.

30 . POKE statement

This statement writes data in memory.

poke statement



Example: *poke* (X, Y)

Stores the value of X in the address indicated by Y. Variables may be declared as *file*.

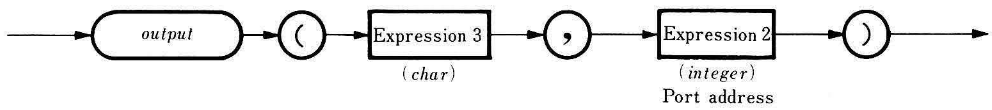
poke ('A', 24576) . . . . Stores ASCII code 65 ($41) corresponding to character A in address 24576 ($6000).

poke ('B', –12288) . . . . . . Stores ASCII code 66 ($42) corresponding to character B in address –12288 ($D000).

31 . OUTPUT statement

This statement outputs data to the specified port.

output statement



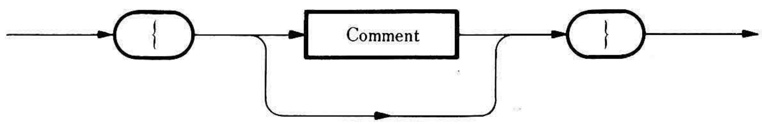
Example : **OUTPUT** (X, A)

Outputs the value of X to the port indicated by A. The value of expression 1 is loaded into the accumulator and the value of expression 2 is loaded into the BC register. Then, machine language instruction OUT (C), A is automatically executed. Expression 1 must be of the *char* type and the codes used are the ASCII codes. Expressions 1 and 2 may be declared as *file.*

32 . Comment statement

This statement outputs a comment.

Comment statement



Example: { AREA OF TRIANGLE }

No { or } symbol can be specified between two other { } symbols. No comment can be specified in any identifier, expression or instruction.

33. Standard function

(1) ODD (< expression >)

The parameter must be an *integer* value and *boolean* result is obtained. This function gives *true* if the parameter is odd, otherwise it gives *false*.

A : = *odd* (5) *true* is assigned to variable A.
A : = *odd* (6) *false* is assigned to variable A.

(2) CHR (< expression >)

The parameter specified in this function must be an *integer* value and a *char* value is obtained as the result.
This function gives the character whose code value is specified in the parameter.

A : = *chr* (80) The character 'P' is assigned to variable A.

(3) ORD (< expression >)

The parameter specified in this function must be a *char* value and an *integer* value is obtained as the result.
This function gives the *integer* value corresponding to the code for the character specified in the parameter.

A : = *ord* ('X') 88 (the code for 'X') is assigned to variable A.

(4) PRED (< expression >)

The parameter specified in this function must be a *char* value and a *char* value is obtained as the result.
This function gives the character which has the same code value as that of the character specified in its parameter, minus 1.

A : = *pred* ('Y') The character 'X' is assigned to variable A.

(5) SUCC (< expression >)

The parameter specified in this function must be a *char* value and a *char* value is obtained as the result.
This function gives the character which has the same code value as that of the character specified in its parameter plus 1.

A : = *succ* ('Y') The character 'Z' is assigned to variable A.

(6) TRUNC (< expression >)

The parameter specified in this function must be a *real* value and an *integer* value is obtained as the result.
This function converts *real* data values into *integer* data values.

A : = *trunc* (3.14) The *integer* value 3 is assigned to variable A.

(7) FLOAT (< expression >)

The parameter specified in this function must be an *integer* value and *real* value is obtained as the result.
This function is the inverse of the *trunc* function; it converts *integer* data values to *real* data values.

A : = *float* (15) *real* number 15.0 is assigned to variable A.

(8) ABS (< expression >)

The result is a *real* value when the value specified in the parameter is *real*; the result is an *integer* value when the value spedified in the parameter is an *integer* value.
This function gives the absolute value of the value specified in the parameter.

A : = *abs* (–3.5) *real* number 3.5 is assigned to variable A.
B : = *abs* (–36.5) *integer* number 36.5 is assigned to variable B.

(9) SQRT (< expression >)

The parameter specified in this function must be a *real* value which is greater than or equal to zero. The result is a *real* value.
This function gives the square root of the value specified in the parameter.

A : = *sqrt* (2.0)   The square root of 2.0 is assigned to variable A.

(10) SIN (< expression >)

The parameter specified in this function must be a *real* value (expressed in radians) and a *real* value is obtained as the result. This function gives the sine of the value specified in the parameter.

To obtain sin 30°, specify
A : = *sin* (30.0∗3.1415927/180.0)

(11) COS (< expression >)

The parameter specified in this function must be a *real* value (in radians) and a *real* value is obtained as the result.

A : = *cos* (200.0∗3.1415927/180.0)
The value of cos 200° is assigned to variable A.

(12) TAN (< expression >)

The parameter specified in this function must be a *real* value (in radians) and a *real* value is obtained as the result.
result.
A : = *tan* (30.0∗3.1415927/180.0)
The value of tan 30° is assigned to variable A.

(13) ARCTAN (< expression >)

The parameter specified in this function must be a *real* value and a *real* value between $-\pi/2$ and $\pi/2$ (in radians) is obtained as the result.

A : =180.0/3.1415927∗ *arctan* (X)
The value of $\tan^{-1}$ X in degrees is assigned to variable A.

(14) EXP (< expression >)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result.
This function gives the value of e^x, where e=2.7182818.

A : = *exp* (1.0)   2.7182818 is assigned to variable A.

(15) LN (< expression >)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result.
This function gives the value of $\log_e X$, where $X \geq 0$.

A : = *ln* (3.0)   1.0986123 is assigned to variable A.

(16) LOG (< expression >)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result.
This function gives the value of $\log_{10} X$, where $X \geq 0$.

A : = *log* (3.0)   0.47712125 is assigned to variable A.

(17) RND (< expression >)

The parameter specified in this function must be a *real* value and a *real* value is obtained as the result.
This function generates pseudo-random numbers between 0.00000001 and 0.99999999.

A : = *rnd* (1.0)	When the value specified as the parameter is larger than 0, the function gives a pseudo-random number.
A : = *rnd* (−1.0)	When the value is 0 or negative, the function generates a pseudo-random number group and gives its initial value.

(18) PEEK (< expression >)

The parameter specified in this function must be an *integer* value and a *char* vlaue is obtained as the result.
This function gives a code (0-255) which corresponds to data stored in the address specified (in decimal) by the parameter.

A : = *peek* (4608) The data code stored in address 4608 is assigned to variable A.

(19) CIN

This function has no parameter, and a *char* value is obtained as the result. This function gives the ASCII code which corresponds to the character in the position on the CRT screen at which the cursor is located.

A : = *cin* The ASCII code of the character displayed at the cursor position is assigned to variable A.

(20) INPUT (< expression >)

The parameter specified in this function must be an *integer* value and a *char* value is obtained as the result.
This function reads data on the port specified by the parameter. For port specification, refer to the explanation of the *output* statement on page 84.
This function executes machine language code $ED78, (i.e. **IN** A, (C)). The value of X is loaded in the BC register and data is read into the accumulator.

A : = *input* (255) Data on port 255 ($FF) is read into variable A.

(21) KEY

This function has no parameter, and a *char* value is obtained as the result. This function gives the ASCII code corresponding to that of the key being pressed. If no key is pressed when this function is executed, the code corresponding to zero is obtained.

A : = *key* The ASCII code corresponding to the key being pressed is assigned to A.

The following statements loop until some key is pressed.

A : = *key;*
while *ord* (A) = 0 **do** A: = *key* ;

(22) CSRH

This function has no parameter, and an *integer* value is obtained as the result. The *integer* value indicates the current location of the cursor on the horizontal axis. The value of this function takes stays within the following ranges for each character display mode:

80-character mode: $0 \leqq csrh \leqq 79$
40-character mode: $0 \leqq csrh \leqq 39$

(23) CSRV

This function has no parameter, and an *integer* value is obtained as the result in the same manner as the *csrh* function. The value indicates the current location of the cursor on the vertical axis and takes stays within the following range for both character modes mentioned above:

$$0 \leqq csrv \leqq 24$$

(24) POSH

This function has no parameter, and an *integer* value is obtained as the result. The *integer* value indicates current location on the horizontal axis of the position pointer in the graphic display area. The value takes stays within the following range:

$$0 \leqq posh \leqq 319$$

(25) POSV

This function has no parameter, and an *integer* value is obtained as the result in the same manner as the *posh* function. The value indicates the current location on the vertical axis of the position pointer in the graphic display area and takes stays within the following range:

$$0 \leqq posv \leqq 199$$

(26) POINT (< expression >, < expression >)

This function has two parameters which must be *integer* value, and an *integer* values is obtained as the result. The value is indicating whether the dot (X, Y) in the graphic display area is set or reset.

Result of the *point* function	Point information
0	Points in both graphic areas 1 and 2 are reset.
1	Only point in graphic area 1 is set.
2	Only point in graphic area 2 is set.
3	Points in both graphic areas 1 and 2 are set.

34 . Standard constant

true *false*	Boolean value

35 . Operator

(1) Integer operators

Operator	Meaning	Example
+	Identity	+A
–	Sign inversion	–B
+	Addition	A+B
–	Subtraction	A–B
∗	Multiplication	A B
div	Division with truncation	A **div** B
mod	Modulus	A **mod** B

Precedence	
1	∗
1	**div**
1	**mod**
2	+
2	–

(2) Real operators

Operator	Meaning	Example
+	Identity	+A
–	Sign inversion	–B
+	Addition	A+B
–	Subtraction	A–B
∗	Multiplication	A∗B
/	Division with truncation	A/B

Precedence	
1	∗
1	/
2	+
2	–

Note: Mixed operations including both integer and real operators are not allowed.

(3) Boolean operators

Operator	Meaning	Example	Precedence
not	Logical NOT	**not** (A=B)	1
and	Logical AND	(A>B) **and** (A>C)	2
or	Logical OR	(A>B) **or** (A>C)	3
xor	Exclusive OR	(A>B) **xor** (A>C)	3

NOT A

Value of A	*true*	*false*
not A	*false*	*true*

A **and** B

Value of A / Value of B	*true*	*false*
true	*true*	*false*
false	*false*	*false*

A **or** B

Value of A / Value of B	*true*	*false*
true	*true*	*true*
false	*true*	*false*

A **xor** B

Value of A / Value of B	*true*	*false*
true	*false*	*true*
true	*true*	*false*

(4) Relational operators

=, <>, <=, >=, < and >. All have equal precedence.

The relational operators may be used for any data types; *integer, real, char* or *boolean*. For *boolean* values, *true* > *false* is always satisfied. Character codes are compared for corresponding *char* type data.

36 . INTEGER and REAL expressions

INTEGER type	REAL type
0	0.0
5	5.0
–135	–135.0
10000	10000.0 or 1E+4

37 . Writing programs by hand

It is recommended that bold faced words such as **procedure**, **begin**, **end** and **var** be underlined (leg., **begin**, **var**).

38 . Indentation

The number of spaces preceding a statement is not prescribed. Use an appropriate number so that the relationship is maintained between **if** and **else**, **begin** and **end**, etc. The preceding spaces do not require any memory spaces.

39 . Statement and Function

Care must be taken when using the following statements and functions since they are similar.

Statement: *output, poke, cout*
Function: *input, peek, cin, key*

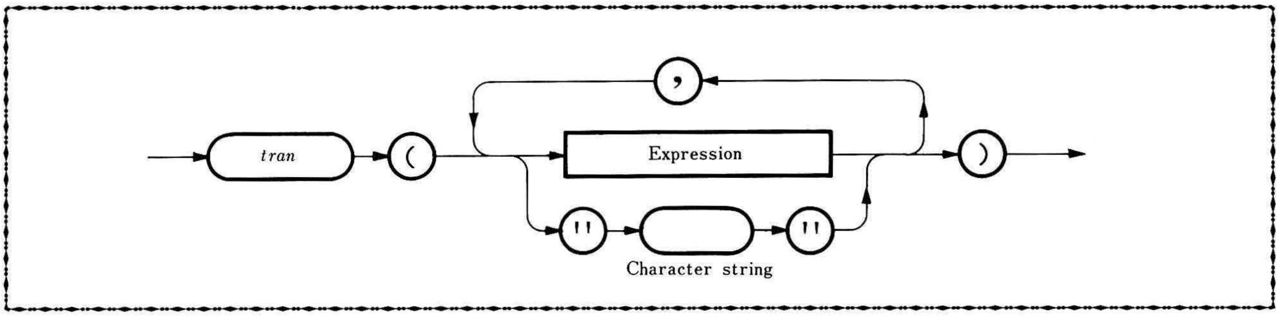
40 . Reserved words

AND	ARRAY	BEGIN	CASE
DIV	DO	DOWNTO	ELSE
END	FILE	FOR	FUNCTION
IF	MOD	NOT	OF
OR	PROCEDURE	REPEAT	THEN
TO	UNTIL	VAR	WHILE
XOR			

41 . Statements for the color control system

1. **TRAN**

   This statement transfers a graphic command to the color control terminal.



2. **REQRT**

   This is a function and it receives a byte of data from the color control terminal.
   This function has no parameter and one character of *char* data is obtained.

3. **SYRET**

   This statement resets the color control terminal and cold starts the system. It has no parameter.

4. **SYRET2**

   This statement resets the color control terminal and waits for a monitor command. It has no parameter.

42 . MUSIC and TEMPO statements

These statements play music. The *tempo* statement specifies the tempo and the *music* statement specifies notes to be played.

tempo (< expression >)

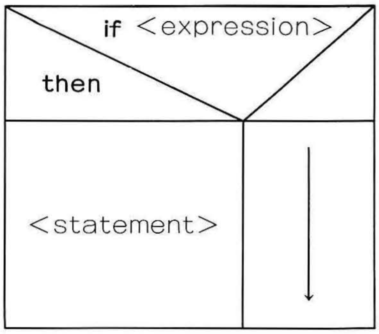
The expression is of the *integer* type and must be in range of 1 ~ 7.

music (< "character string" > | < *char* expression >)

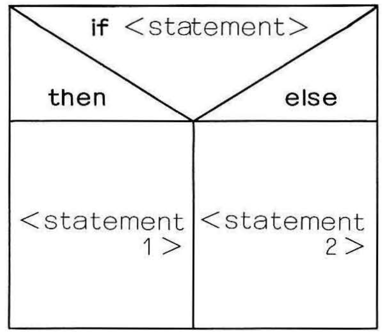
Notes are specified with the character string or the *char* expression.

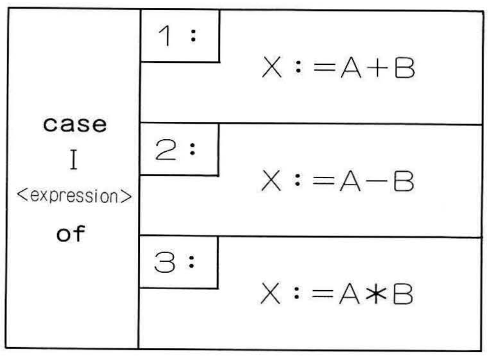
43 . NS chart

if < expression > **then** < statement >



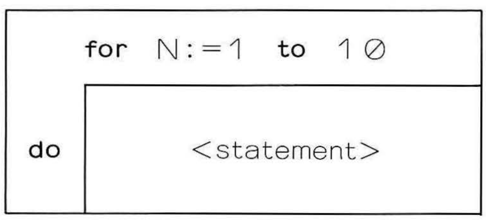
if < expression > **then** < statement 1 >
else < statement 2 >

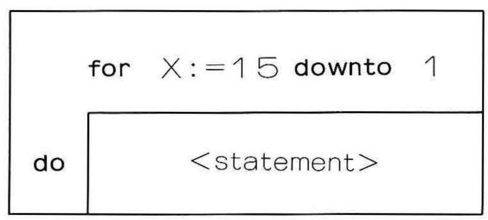




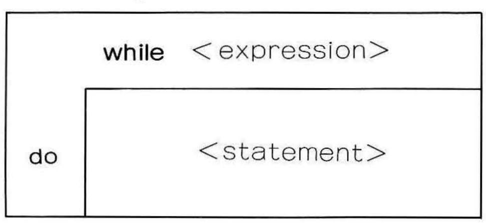
case I **of** 1 : X : =A+B ;
<expres-sion> 2 : X : =A−B ;
3 : X : =A B

for < control variable > : = < initial value > **to** < final value > **do** < statement >
for < control variable > : = < initial value > **downto** < final value > **do** < statement >

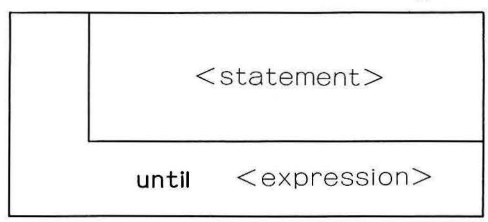




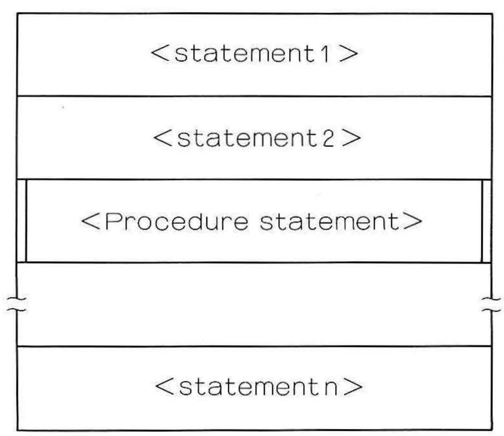
while < expression > **do** < statement >



repeat < statement > **until** < expression >



Compound statement **begin** < statement 1 > ; < statement 2 > ; < procedure call > ; ...... ; < statement n > **end**



APPENDIX

ASCII code table

A table of hexadecimal ASCII codes is shown in FIGURE 2.22 of the Owner's Manual.

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
0	NULL	26		52	4	78	N	104	h
1	↓	27		53	5	79	O	105	i
2	↑	28		54	6	80	P	106	j
3	→	29		55	7	81	Q	107	k
4	←	30		56	8	82	R	108	l
5	HOME	31	▒	57	9	83	S	109	m
6	CLR	32		58	:	84	T	110	n
7	DEL	33	!	59	;	85	U	111	o
8	INST	34	"	60	<	86	V	112	p
9	GRPH	35	#	61	=	87	W	113	q
10	SFT LOCK	36	$	62	>	88	X	114	r
11		37	%	63	?	89	Y	115	s
12	RVS	38	&	64	@	90	Z	116	t
13		39	'	65	A	91	[	117	u
14	L SCRIPT	40	(	66	B	92	\	118	v
15	RVS CANCEL	41	)	67	C	93	]	119	w
16		42	*	68	D	94	^	120	x
17		43	+	69	E	95	‾	121	y
18		44	,	70	F	96	`	122	z
19		45	−	71	G	97	a	123	{
20		46	.	72	H	98	b	124	|
21		47	/	73	I	99	c	125	}
22		48	0	74	J	100	d	126	~
23		49	1	75	K	101	e	127	↴
24		50	2	76	L	102	f		
25		51	3	77	M	103	g		

CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER	CODE	CHARACTER
128		154		180	4	206	N	232	h
129	↓	155		181	5	207	O	233	i
130	↑	156		182	6	208	P	234	j
131	→	157		183	7	209	Q	235	k
132	←	158		184	8	210	R	236	l
133	♠	159		185	9	211	S	237	m
134	♥	160		186	:	212	T	238	n
135	♦	161	!	187	;	213	U	239	o
136	♣	162	"	188	<	214	V	240	p
137		163	#	189	=	215	W	241	q
138		164	$	190	>	216	X	242	r
139		165	%	191	?	217	Y	243	s
140		166	&	192	@	218	Z	244	t
141		167	'	193	A	219	[	245	u
142		168	(	194	B	220	\	246	v
143		169	)	195	C	221	]	247	w
144		170	*	196	D	222	^	248	x
145	¥	171	+	197	E	223	_	249	y
146	£	172	,	198	F	224	`	250	z
147	●	173	-	199	G	225	a	251	{
148	○	174	.	200	H	226	b	252	|
149		175	/	201	I	227	c	253	}
150		176	0	202	J	228	d	254	~
151		177	1	203	K	229	e	255	π
152		178	2	204	L	230	f		
153		179	3	205	M	231	g		

Decimal/Hexadecimal conversion table

Decimal	Hexa-decimal	Decimal	Hexa-decimal	Decimal	Hexa-decimal	Decimal	Hexa-decimal	Decimal	Hexa-decimal	Decimal	Hexa-decimal
0	00	48	30	96	60	144	90	192	C0	240	F0
1	01	49	31	97	61	145	91	193	C1	241	F1
2	02	50	32	98	62	146	92	194	C2	242	F2
3	03	51	32	99	63	147	93	195	C3	243	F3
4	04	52	34	100	64	148	94	196	C4	244	F4
5	05	53	35	101	65	149	95	197	C5	245	F5
6	06	54	36	102	66	150	96	198	C6	246	F6
7	07	55	37	103	67	151	97	199	C7	247	F7
8	08	56	38	104	68	152	98	200	C8	248	F8
9	09	57	39	105	69	153	99	201	C9	249	F9
10	0A	58	3A	106	6A	154	9A	202	CA	250	FA
11	0B	59	3B	107	6B	155	9B	203	CB	251	FB
12	0C	60	3C	108	6C	156	9C	204	CC	252	FC
13	0D	61	3D	109	6D	157	9D	205	CD	253	FD
14	0E	62	3E	110	6E	158	9E	206	CE	254	FE
15	0F	63	3F	111	6F	159	9F	207	CF	255	FF
16	10	64	40	112	70	160	A0	208	D0		
17	11	65	41	113	71	161	A1	209	D1		
18	12	66	42	114	72	162	A2	210	D2		
19	13	67	43	115	73	163	A3	211	D3		
20	14	68	44	116	74	164	A4	212	D4		
21	15	69	45	117	75	165	A5	213	D5		
22	16	70	46	118	76	166	A6	214	D6		
23	17	71	47	119	77	167	A7	215	D7		
24	18	72	48	120	78	168	A8	216	D8		
25	19	73	49	121	79	169	A9	217	D9		
26	1A	74	4A	122	7A	170	AA	218	DA		
27	1B	75	4B	123	7B	171	AB	219	DB		
28	1C	76	4C	124	7C	172	AC	220	DC		
29	1D	77	4D	125	7D	173	AD	221	DD		
30	1E	78	4E	126	7E	174	AE	222	DE		
31	1F	79	4F	127	7F	175	AF	223	DF		
32	20	80	50	128	80	176	B0	224	E0		
33	21	81	51	129	81	177	B1	225	E1		
34	22	82	52	130	82	178	B2	226	E2		
35	23	83	53	131	83	179	B3	227	E3		
36	24	84	54	132	84	180	B4	228	E4		
37	25	85	55	133	85	181	B5	229	E5		
38	26	86	56	134	86	182	B6	230	E6		
39	27	87	57	135	87	183	B7	231	E7		
40	28	88	58	136	88	184	B8	232	E8		
41	29	89	59	137	89	185	B9	233	E9		
42	2A	90	5A	138	8A	186	BA	234	EA		
43	2B	91	5B	139	8B	187	BB	235	EB		
44	2C	92	5C	140	8C	188	BC	236	EC		
45	2D	93	5D	141	8D	189	BD	237	ED		
46	2E	94	5E	142	8E	190	BE	238	EE		
47	2F	95	5F	143	8F	191	BF	239	EF		

Error message table

Error code	Description
1	The program is not completed or . is omitted.
2	An identifier is declared twice.
3	: is omitted or a character other than : is specified in a place where : should be specified.
4	Type specified is not allowed.
5	Other than identifier is specified in a place where an identifier should be specified.
6	An identifier is too long.
7	OF is omitted.
8	) or , is omitted.
9	(is omitted.
10	[or , is omitted.
11	] is omitted.
12	Other than an integer is specified in a place where an integer should be specified.
13	An array element is too large or data is out of the declared range.
14	; is omitted.
15	, is omitted.
16	A READ or WRITE statement includes mixed specifications of FILE type variables and other types of variables.
17	An incorrect type of value is assigned to a variable.
18	; or END is omitted.
19	THEN is omitted.
20	Other than a BOOLEAN type variable is specified in a place where a BOOLEAN type variable should be specified.
21	DO is omitted.
22	:= is omitted.
23	TO or DOWNTO is omitted.
24	UNTIL is omitted.
25	Other than a variable, function or constant is specified in the place where one of these should be specified.
26	More than one character is included between single quotation marks.
27	A undeclared identifier is used.
28	Other than a procedure identifier is specified where one should be specified.
29	Parameter mismatch or array dimension mismatch.
30	BEGIN is omitted.
31	Other than a digit is specified where one should be specified.
32	Other than REAL is specified where REAL should be specified.
33	Other than, or CR is keyed in where either of these two should be keyed in.
34	WRITE error or break during WRITE execution.
35	READ error or break during READ execution.

Error code	Description
36	The number of digits of data exceeds the specified number of digits in the WRITE statement.
37	$ is omitted.
38	Other than hexadecimal data is specified where hexadecimal data should be specified.
39	Insufficient memory capacity
40	Command error
41	", ', { or } is omitted.
42	Printer is OFF or is not connected.
43	Printer out of paper
44	Printer mechanical trouble
45	Unallowed symbol is specified.
46	CLOSE is omitted.
47	An unopened file was referenced. FNAME is omitted.
48	
49	
50	
51	
52	
53	
54	
55	

A program is checked for syntax before execution begins. If an error is found, the following message will be output.

∗ **Err** < error code > ∗ **Line** < line number >

∗ **Err** < error code > ∗ **Run** ∗ indicates a non-syntactical error: error code 36 is one of non-syntactical errors.

Note: It may happen that no error can be found on the indicated line even though an error message is output. A possible cause is erroneous loading of the program. In such cases, display the program list around the indicated line. Position the cursor on the line in which the error is indicated and perform a carriage return to reload the program.

PASCAL SB-4515 specifications

System	Cassette tape base interpreter; interpreter; Monitor SB-1511
Size	Approx. 20K bytes
Required RAM capacity	64K bytes
Cold start address	$1300
Warm start address	$1301
Error messages	47 messages
Data types	*integer* *real* *char* *boolean*
Data range	
INTEGER data	–32767 ~ +32767 (2 byte data, 2's complement)
REAL data	±0.27105055E–19 ~ ±0.92233720E+19
CHAR data	One character (corresponding to codes 0~255)
BOOLEAN data	*true* and *false* (*true*> *false*)
Number of array dimensions	Up to n dimensions
Range of array index	Varies according to data type and memory size
Identifier length	Up to 32 characters
Integer operators	∗, **div**, **mod**, +,
Real operators	∗, /, +, –
Logical operators	**not**, **and**, **or**, **xor**
Relational operators	=, <>, <=, >=, <, >
Standard functions	
ODD	Checking whether odd or even. *boolean* ← *integer*
CHR	*char* ← *integer*
ORD	*integer* ← *char*
PRED	Preceding character
SUCC	Following character
TRUNC	*integer* ← *real*
FLOAT	*real* ← *integer*
ABS	Absolute value of *integer* or *real* data
SQRT	Square root
SIN	sinX
COS	cosX
TAN	tanX
ARCTAN	$\tan^{-1}X$
EXP	e^x
LN	$\log_e X$
LOG	$\log_{10} X$
RND	Random number
PEEK	Read-out from memory
CIN	Read in character at cursor position
INPUT	Read in from port
KEY	Read in from keyboard
REQTR	Receive one byte of data from the color control system
CSRH	Current location of the cursor on the horizontal axis
CSRV	Current location of the cursor on the vertical axis.

POSH	Current location of the position pointer on the X-axis
POSV	Current location of the position pointer on the Y-axis
POINT	Determine whether specified dots are set or reset

Statements

Assignment statement	Variable: =< expression >
Compound statement	**begin** < statement 1 >, < statement 2 >, ... , < statement n > **end;**
IF statement	Conditional statement (including **else**)
CASE statement	Selective execution
WHILE statement	Repetition
REPEAT statement	Repetition
FOR statement	Repetition (including either **to** or **downto**)
WRITE statement	Data output
READ statement	Data input
FNAME statement	Defines the file name of a data file and opens it.
CLOSE statement	Closes the data file which was opened by the *fname* statement.
GRAPH statement	Specifies the graphic input/output area, clears or fills graphic area.
GSET statement	Sets a dot in the specified position in a graphic area.
GRSET statement	Resets a dot in the specified position in a graphic area.
LINE statement	Draws lines connecting positions specified by operands
BLINE statement	Draws black lines connecting positions specified by operands.
POSITION statement	Sets the location of the position pointer in a graphic area.
PATTERN statement	Draws the dot pattern specified by operand in a graphic area.
RANGE statement	Sets the scrolling area, number of characters/line or reverse/normal.
CURSOR statement	Moves the cursor to any position on the screen.
FKEY statement	Defines a function for any of the definable function keys.
CALL statement	User subroutine call
COUT statement	Outputs a character to the cursor position
POKE statement	Writes data into memory
OUTPUT statement	Outputs data to the specified port
MUSIC statement	Plays music (used with the *tempo* statement).
TEMPO statement	Specifies the tempo.
COPY statement	Makes a copy of the character display or graphic display.
IMAGE statement	Draws the dot pattern specified in the operand on the printer.
TRAN statement	Transfers a graphic command to the color terminal.
SYRET statement	Resets the color terminal and cold starts the system.
SYRET2 statement	Resets the color terminal and waits for a monitor command.

Others

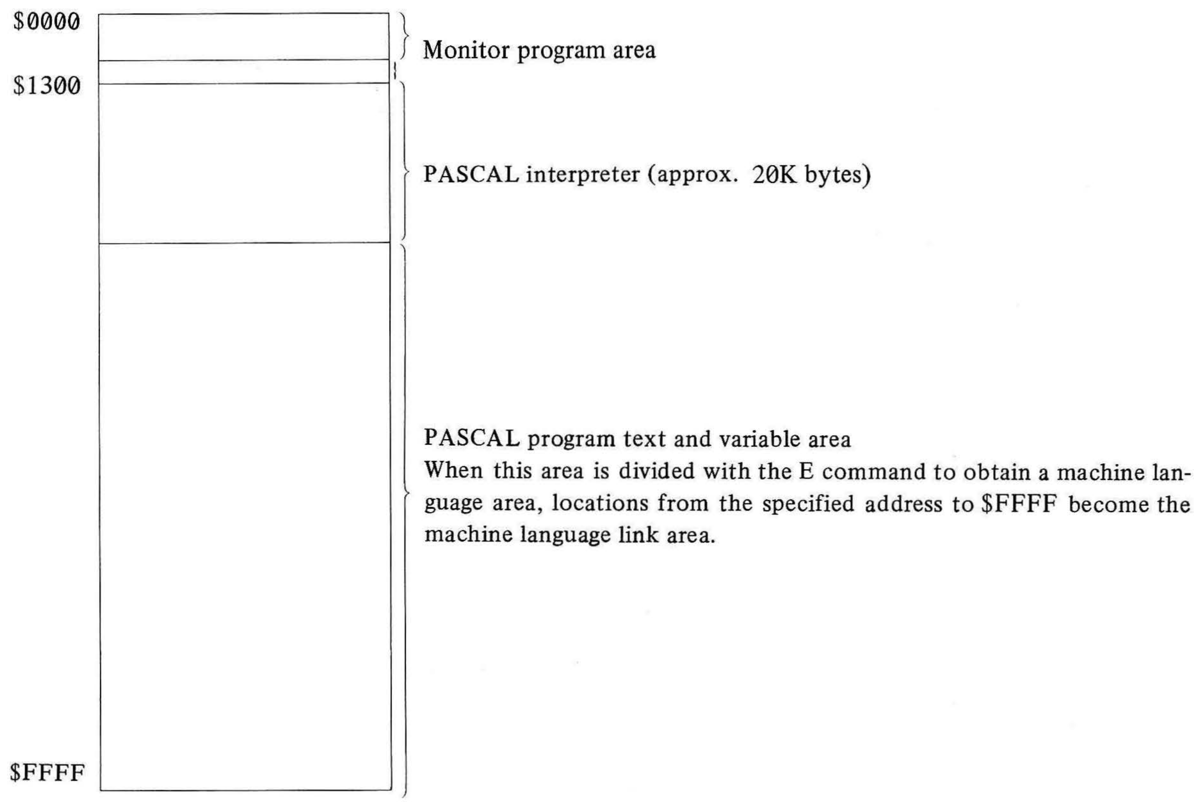
(1)	A file identifier can include a maximum of 16 significant characters.
(2)	Statement numbers are automatically assigned by the system.
(3)	Recursive call capability.

Differences between the SB-4515 and standard PASCAL

1. No procedure or function can be declared within another procedure or function declaration.
2. Structured type data cannot be used.
3. Only value parameters can be used.

Memory map

The memory map is as shown below when the PASCAL interpreter is loaded in a system.



(Hexadecimal address)

PASCAL SB-4515 configuration

PASCAL SB-4515 is roughly divided into three sections; program control flows as shown in Figure A.1.

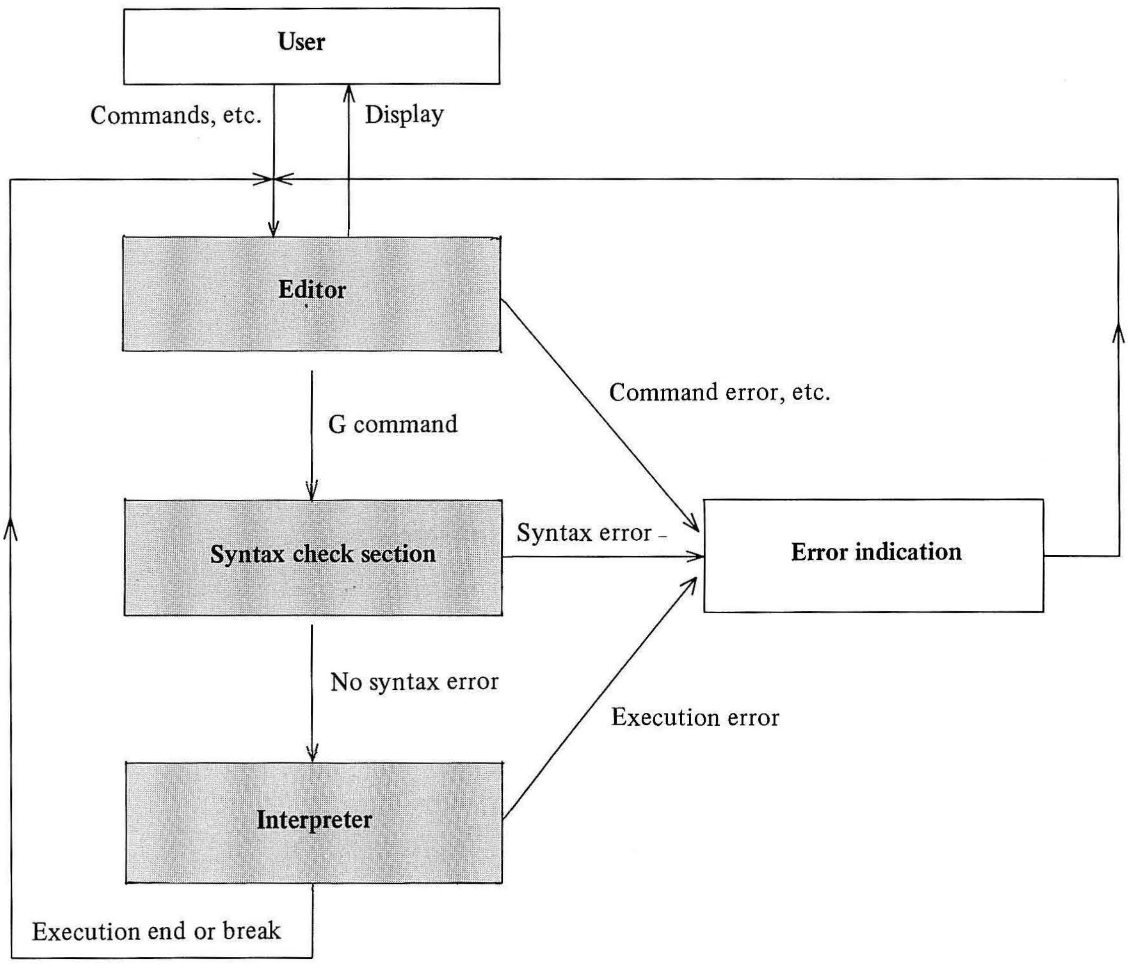
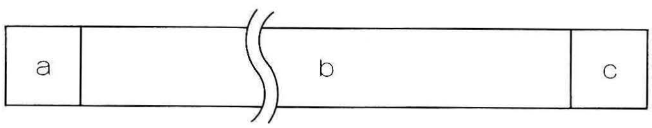


Figure A.1 SB-4515 CONTROL FLOW



1. EDITING SECTION

The editor is primarily used for executing editing commands and generating a source programs.

Each line of a source program is converted into an intermediate code line when it is loaded. (See Figure A.2.) One intermediate code line corresponds to one source program line. Line numbers are omitted in intermediate codes.

a : Number of spaces for indentation (1 byte)
b : Intermediate codes and identifiers (n bytes : n is indefinite.)
c : ØDH indicating line end (1 byte)

Source program lines are converted into the form shown above. Line number data is omitted.

Figure A.2

Figure A.3 shows a map of the memory during editing of a source program. The line numbers only control the contents of the line pointer (LP), which a line may be inserted or from which a line may be deleted. The P and H commands reconvert intermediate codes into source program lines for display.

Standard functions are assigned to intermediate codes. For functions which operate on parameters, such as COS (A), one intermediate code is assigned to COS (and others are assigned to A and). Therefore, COS□ (A) is not handled as the COS function, but is handled as two identifiers COS □ (A) are converted into intermediate codes as shown in Figure A.4.

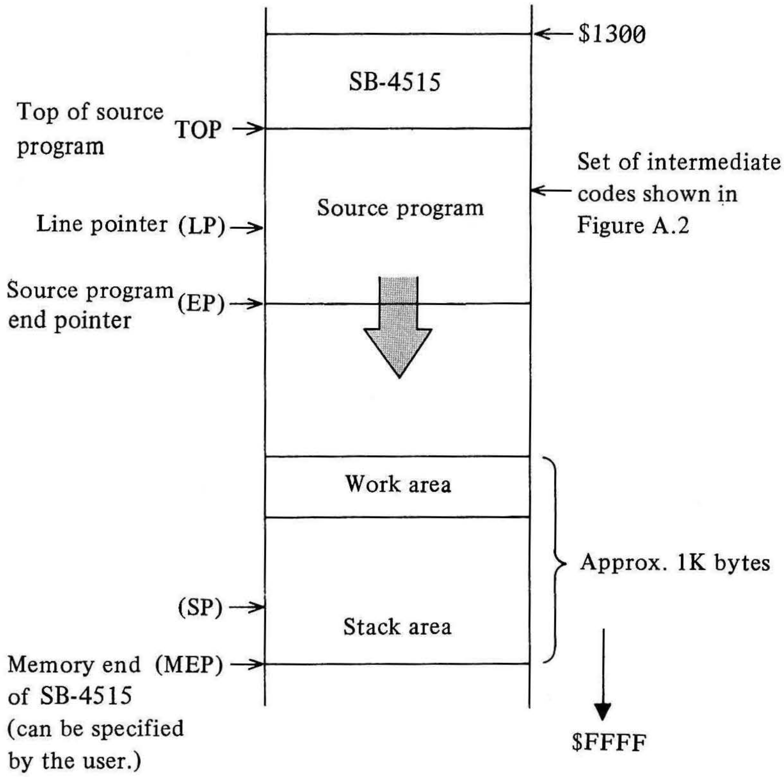


Figure A.3 Memory Map during Editing

Example 1:

· · · · COS (A) · · · · ·

· · · · ·	a_0	A	a_1	a_2	a_3	a_4	a_5	· · · · ·

a_0 : Intermediate code for COS (.
A : Identifier . . . . ASCII code
a_1 : Intermediate code for the identifier
a_2 : Flag data representing data type
a_3, a_4 : Pointer data indicating static address
a_5 : Intermediate code for)

Example 2:

· · · · COS□ (A) · · · · ·

□ represents a space.

· · · · ·	C	O	S	b_1	b_2	b_3	b_4	b_5	A	b_6	b_7	b_8	b_9	b_A	· · · · ·

COS, A : Identifiers . . . . ASCII codes
b_1, b_6 : Intermediate codes for the identifiers
b_2, b_7 : Flag data representing data types
b_3, b_4, b_8, b_9 : Pointer data indicating static addresses
b_5 : Intermediate code for (
b_A : Intermediate code for)

When the above two intermediate code lines are displayed with the P command, both source program lines are displayed in the form . . . . COS (A) . . . .

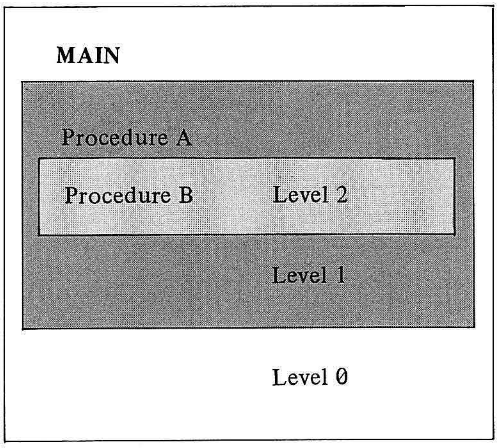
Figure A.4

2. SYNTAX CHECK SECTION

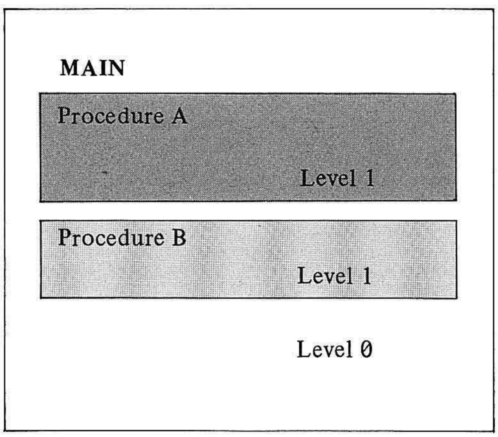
Figure A.5 shows difference in block structure between standard PASCAL and SB-4515. A lexical level of up to 1 is allowed for the SB-4515.

Figure A.6 shows a map of the memory configuration during a syntax check. The syntax check section determines variable types and static addresses, analyzes the structure of user programs and completes the intermediate code data section.

Figure A.7 shows an example of identifier analysis and Figure A.8 an example of program structure analysis.



(a) Standard PASCAL block structure



(b) SB-4515 block structure

Figure A.5

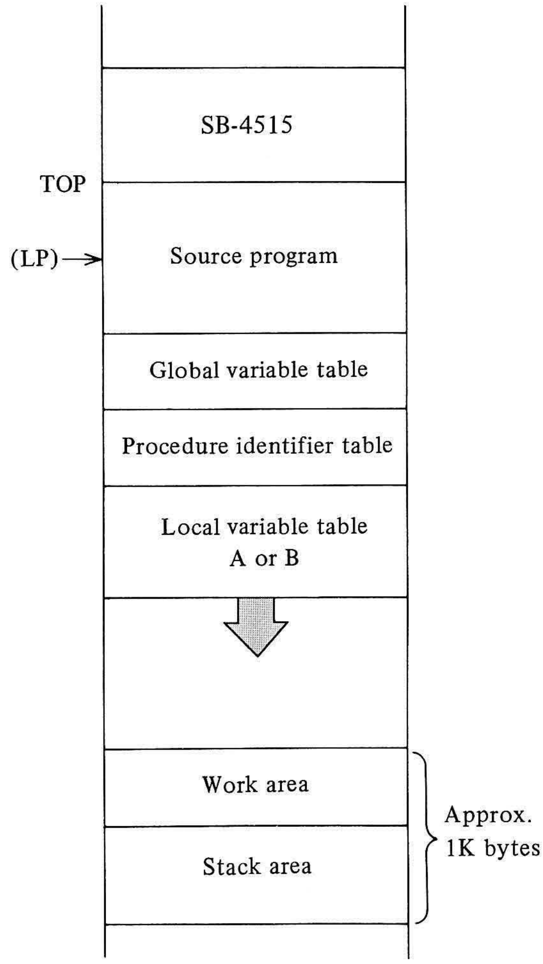
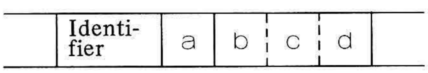


Figure A.6 Memory map during syntax check



Identifier : ASCII code of up to 32 characters

a : Code distinguishing
- procedure
- function
- global variable
- local variable

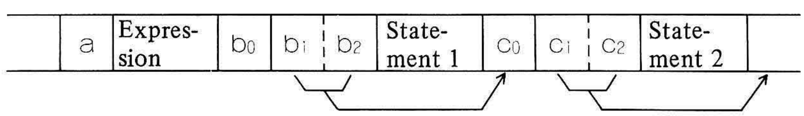
b : Flag indicating type

c, d : Address data

Figure A.7

The following is the intermediate code for

if <expression> **then** <statement 1>
else <statement 2>.



a : Code representing **if**

b_0 : Code representing **then**

b_1, b_2 : Pointer indicating the address following statement 1

c_0 : Code representing **else**

c_1, c_2 : Pointer indicating the address following statement 2

Figure A.8

3. INTERPRETER SECTION

The interpreter section consists of the syntax analyzing section and the virtual stack machine. The virtual stack machine cannot directly execute intermediate codes which are arranged in the same order as the source program; The syntax analyzing section interprets the intermediate codes to control the virtual stack machine.

Figure A. 9 Shows a map of the memory configuration during execution of the interpreter section. Note that two stacks, S and W, are used. The S stack is mainly used by the syntax analyzing section and the W stack by the virtual stack machine.

Figure A. 9 shows the state when part of the statements of a user defined function have been executed. The stack buffer stores the S stack contents for the main program at the time the function is called.

Figure A. 10 shows the stack buffer structure. Area A stores the address and the number of bytes of data transferred from the S stack, and area B stores the contents of various pointers (NP, RB, etc.). Area C stores the S stack contents.

This data is returned to its original locations when control is returned to the main program.

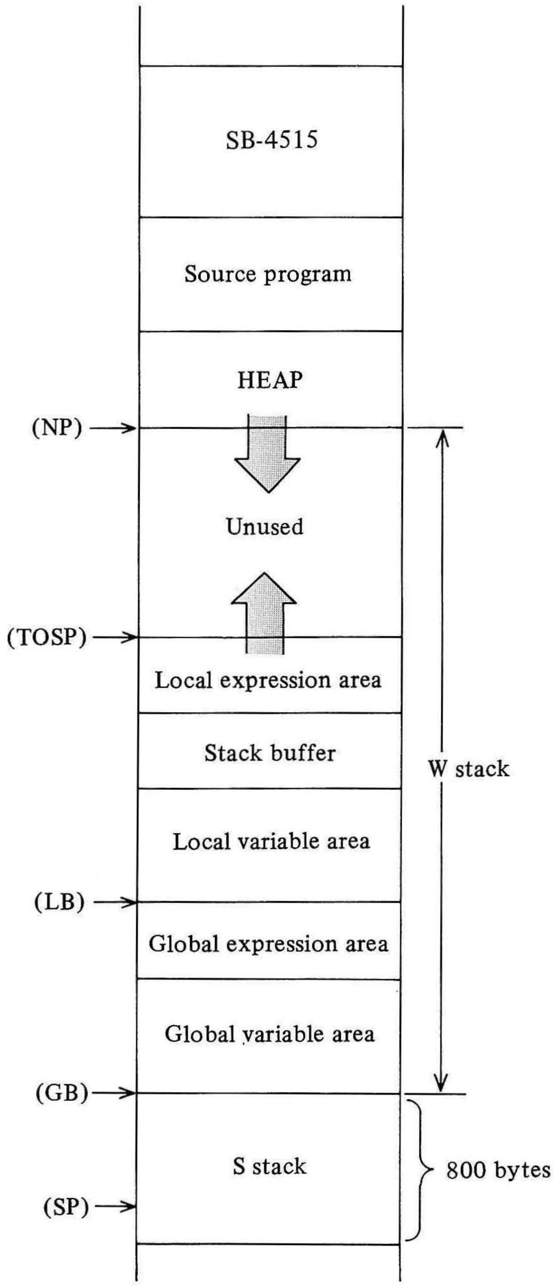


Figure A. 9

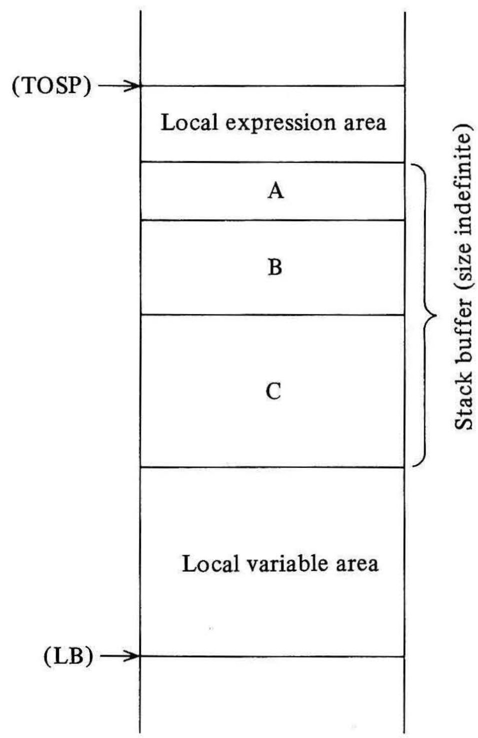


Figure A. 10

1. The address of A is stored in the S stack.
The remainder is : = B∗C.

2. : = is stored in the S stack.
The remainder is B∗C.

3. TOSP ← TOSP-1
The value of B is stored in the W stack.
The remainder is C.

4. ∗ is stored in the S stack.
The remainder is C.

5. TOSP ← TOSP-1
The value of C is stored in the W stack.
There is no remainder.

6. The first element of the S stack is read.
Since it is ∗,
TOSP ← TSOP+1
(TOSP) ← (TOSP) ∗ (TOSP-1)

7. The next element of the S stack is read. Since it is : =, the following element of the S stack (address of A) is also read and the data of (TOSP) is stored in the address of A. Then, TOSP ← TOSP+1.

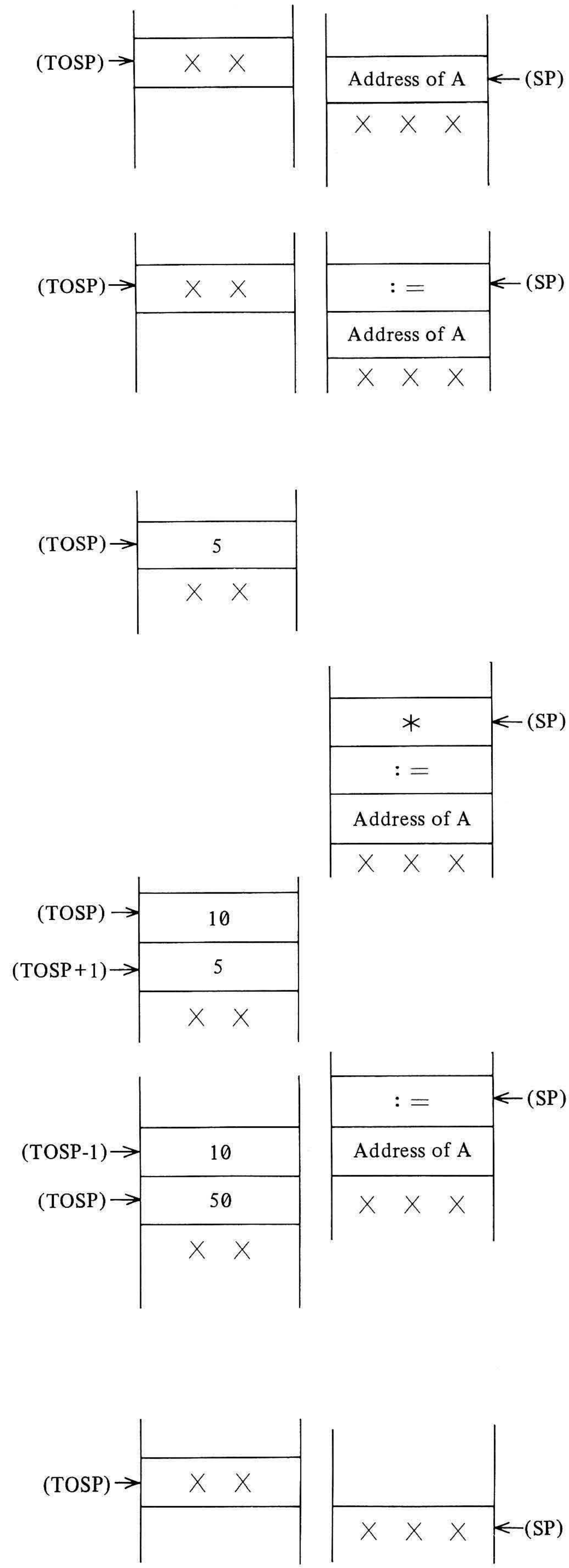


Figure A.11 Execution of A : =B C where B is assigned 5 and C assigned 1.

Figure A.12 shows data formats in the W stack and HEAP area. In the W stack, each data element consists of a type flag (1 byte) and a data section.

For arrays, data sections are stored in the HEAP area and only array pointer are stored in the W stack.

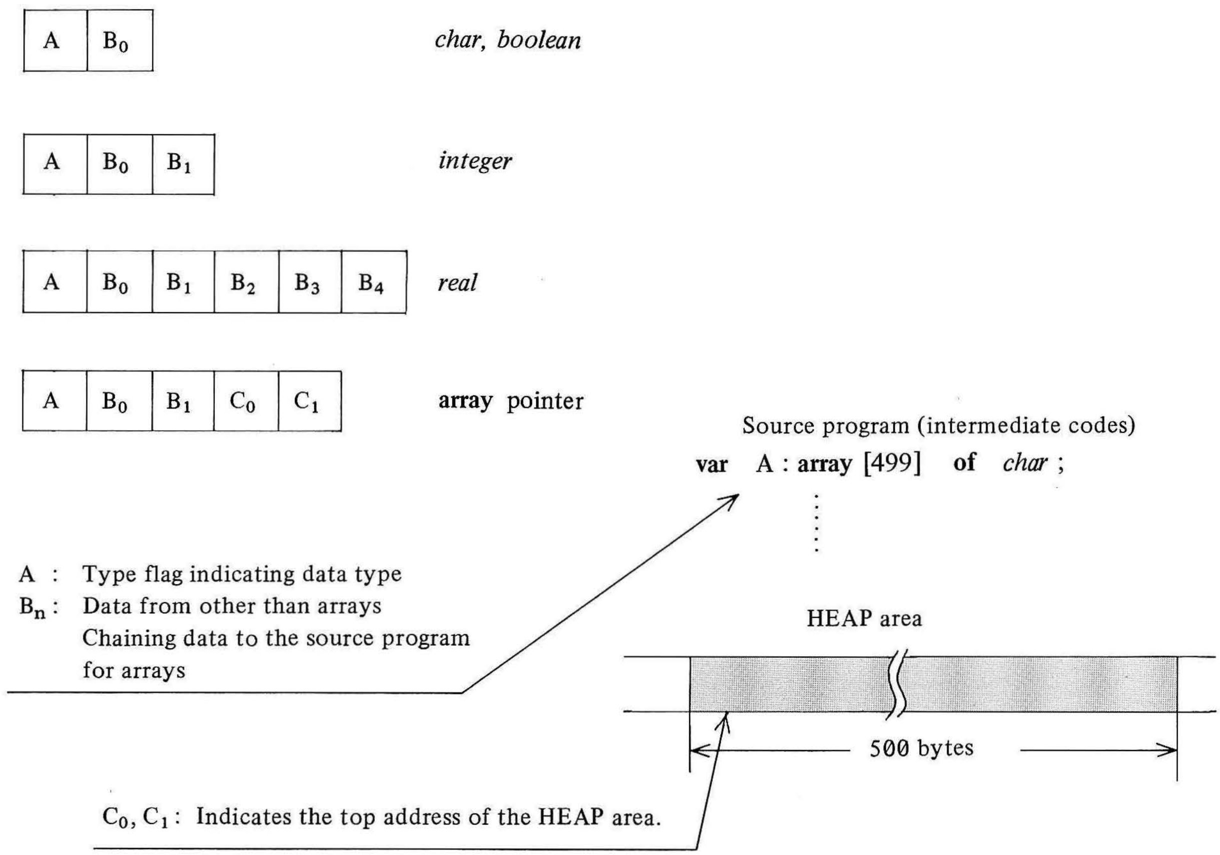


Figure A.12

File declaration

File declaration supported by the SB-4515 only declares the memory area used for transfer of data to/from the cassette tape. This is different file declaration supported by standard PASCAL.

Reference

(1) K. Jensen and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, 1974.
(2) N. Wirth, Algorithms + Data Structures = Programs, Prentice-Hall, 1976.

Sample Program

(1) Hanoi Tower (Application Program 1)

This is the program list for the first application program stored on the PASCAL Applications Tape. The explanations in Chapter 2 are based on this program.

```
{             SAMPLE PROGRAM                 }
{       MASSAGE & HANOI TOWER                }
{}
var A,N,X,Y,CX,CY:integer;
    DISKNO,DISKNUMBER,TOWER1,TOWER2,TOWER3,TOWERNUMBER:integer;
    B,C:char;
procedure TITLE;{ ............................... MESSAGE }
begin
    DELAY(1000);
    write("▣⇩⇩⇩⇩⇩⇩⇦⇦⇦⇦⇦⇦⇦⇦⇦⇦⇦⇦⇦⇦");
    write("┌┬┐⇩⇦⇦|⇩⇦┴");
    write("⇦⇧⇧┬ ┬⇩⇦⇦⇦├─┤⇩⇦⇦⇦┴ ┴");
    writeln("⇧⇧┌─┐⇩⇦⇦⇦├┤⇩⇦⇦└─┘");
    DELAY(500);
    writeln();
    writeln("  ┌──┐  ┌──┐  ┌──┐  ┌──┐  ┌──┐  ┌ ");
    writeln("  |  |  |  |  |     |     |  |  |");
    writeln("  ├──┘  ├──┤  └──┐  |     ├──┤  |");
    writeln("  |     |  |     |  |     |  |  |");
    writeln("  ┴     ┴  ┴  └──┘  └──┘  ┴  ┴  └──▣");
  end;
procedure STARWRITE;{ ......................... STARRY SKY }
  var N,POSITION:integer;
  begin
    for N:=1 to 150 do
      begin
        cursor(trunc(rnd(1.0)*40.0),trunc(rnd(1.0)*24.0));
        cout(chr(46))
      end
  end;
procedure UFOMOVE;{ UFO }
  var N:integer;UFO:char;
  begin
    write("▣");
    for N:=1 to 39 do write("⇦");
    for N:=1 to 4 do write("⇩");
    UFO:=chr(64);
    write(UFO:1);
    for N:=1 to 39 do
      begin
        DELAY(100);
        write("⇦.⇦⇦",UFO:1)
      end;
    write("⇦ ⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩⇩")
  end;
procedure DELAY(D:integer);
  var N:integer;
  begin
    for N:=0 to D do
  end;
procedure CPS;
  begin
    CX:=csrh;CY:=csrv
  end;
```

```
procedure CMOVE(CH:char);
  begin
    cout(CH);if CX<39 then CX:=CX+1;cursor(CX,CY)
  end;
procedure DOW(N:integer);
  begin
    CPS;CY:=CY+N+1;cursor(CX,CY)
  end;
procedure UP(N:integer);
  begin
    CPS;CY:=CY-N-1;cursor(CX,CY)
  end;
procedure RIGHT(N:integer);
  begin
    CPS;CX:=CX+N+1;cursor(CX,CY)
  end;
procedure LEFT(N:integer);
  begin
    CPS;CX:=CX-N-1;cursor(CX,CY)
  end;
procedure START;
  var N:integer;
  begin
    repeat
      begin
        write("▣");
        RIGHT(9);
        writeln("♀◆◆   HANOI TOWER   ◆◆");
        DOW(5);
        for N:=0 to 8 do
          begin
            RIGHT(6);
            write("I");
            RIGHT(11);
            write("I");
            RIGHT(11);
            writeln("I")
          end;
        for N:=1 to 40 do write("╨");
        writeln();
        RIGHT(2);
        writeln("  How many disks");
        RIGHT(2);write("  do you want to move (3-6) ");
        read(DISKNUMBER)
      end
    until(DISKNUMBER<7)and(DISKNUMBER>2);
    TOWER1:=0;
    TOWER2:=0;
    TOWER3:=0;
    for DISKNO:=DISKNUMBER downto 1 do DISKW(DISKNO,1)
  end;
procedure DISKW(DISKNO,TOWERNUMBER:integer);
  var N:integer;
  begin
    if TOWERNUMBER=1 then
      begin
        write("▣");
        A:=15-TOWER1;
        DOW(A);
        A:=6-DISKNO;
```

```
        RIGHT(A);
        for N:=1 to DISKNO do CMOVE('▒');
        RIGHT(0);
        for N:=0 to DISKNO-1 do CMOVE('▒');
        TOWER1:=TOWER1+1
      end
    else if TOWERNUMBER=2 then
           begin
             write("◘");
             A:=15-TOWER2;
             DOW(A);
             A:=19-DISKNO;
             RIGHT(A);
             for N:=1 to DISKNO do CMOVE('▒');
             RIGHT(0);
             for N:=0 to DISKNO-1 do CMOVE('▒');
             TOWER2:=TOWER2+1
           end
         else begin
                write("◘");
                A:=15-TOWER3;
                DOW(A);
                A:=32-DISKNO;
                RIGHT(A);
                for N:=1 to DISKNO do CMOVE('▒');
                RIGHT(0);
                for N:=0 to DISKNO-1 do CMOVE('▒');
                TOWER3:=TOWER3+1
              end;
    DELAY(500)
  end;
procedure DISKD(TOWERNUMBER:integer);
  var N:integer;
  begin
    if TOWERNUMBER=1 then
      begin
        write("◘");
        A:=16-TOWER1;
        DOW(A);
        RIGHT(0);
        for N:=1 to 6 do CMOVE(' ');
        RIGHT(0);
        for N:=0 to 5 do CMOVE(' ');
        TOWER1:=TOWER1-1
      end
    else if TOWERNUMBER=2 then
           begin
             write("◘");
             A:=16-TOWER2;
             DOW(A);
             RIGHT(13);
             for N:=1 to 6 do CMOVE(' ');
             RIGHT(0);
             for N:=0 to 5 do CMOVE(' ');
             TOWER2:=TOWER2-1
           end
         else if TOWERNUMBER=3 then
                begin
                  write("◘");
                  A:=16-TOWER3;
```

```
                  DOW(A);
                  RIGHT(26);
                  for N:=1 to 6 do CMOVE(' ');
                  RIGHT(0);
                  for N:=0 to 5 do CMOVE(' ');
                  TOWER3:=TOWER3-1
                end
  end;
procedure DISKMOV(DISKNUMBER,T1,T2,T3:integer);
  begin
    if DISKNUMBER<>0 then
      begin
        DISKMOV(DISKNUMBER-1,T1,T3,T2);
        DISKD(T1);
        DISKW(DISKNUMBER,T3);
        DISKMOV(DISKNUMBER-1,T2,T1,T3)
      end
  end;
begin { ......................................MAIN PROGRAM }
  range(C,40);
  STARWRITE;
  DELAY(2000);
  TITLE;
  UFOMOVE;
  writeln("⇩⇩⇩                       SHARP CORPORATION⇩⇩⇩⇩");
  write("           [ Press a key ]");
  B:=key;
  while ord(B)=0 do B:=key;
  repeat
    START;
    DISKMOV(DISKNUMBER,1,2,3);
    DOW(6);
    write("▣");DOW(21);RIGHT(5);
    write("   Try again (Y OR N)");
    read(C)
  until 'Y'<>C
end.
```

(2) Eight Queens (Application Program 2)

The following sample program arranges 8 queens on a chessboard so that no queen can take any other. There are 92 solutions. This procedure is often used as an example of recursive programming. **procedure** ARYWRITE calls itself.

This program is the second section of the PASCAL Applications Tape.

```
{ Eight Queens }
var A,X:array[7]of integer;
    B,C:array[14]of integer;
    D,P:integer;
procedure CLEAR;
  var N:integer;
  begin
    for N:=0 to 7 do A[N]:=1;
    for N:=0 to 14 do B[N]:=1;
    for N:=0 to 14 do C[N]:=1;
    P:=0
  end;
procedure ARYWRITE;
  var Z:integer;
  begin
    Z:=0;
    repeat
      if((A[Z]+B[P-Z+7]+C[P+Z])=3)then
        begin
          X[P]:=Z;
          A[Z]:=0;
          B[P-Z+7]:=0;
          C[P+Z]:=0;
          P:=P+1;
          if P=8 then DATAOUT
            else ARYWRITE;{....................A RECURSIVE CALL }
          P:=P-1;
          A[Z]:=1;
          B[P-Z+7]:=1;
          C[P+Z]:=1
        end;
      Z:=Z+1
    until Z=8
  end;
procedure BOARD;
  var E,M,N:integer;
  begin
    D:=0;
    writeln("@↓↓       ◆◆ EIGHT QUEENS ◆◆↓↓↓");
    write("       ┌");
    for M:=0 to 6 do write("─┬");
    writeln("─┐");
    for E:=0 to 7 do
      begin
        write("       │");
        for·M:=0 to 7 do write(" │");
        writeln();
        if E<>7 then
          begin
            write("       ├");
            for M:=0 to 6 do write("─┼");
            write("─┤");
            writeln()
```

```
            end
          else begin write("      └");
                 for M:=0 to 6 do write("─┴");
                 writeln("─┘")
               end
        end
    end;
procedure DATAOUT;
    var F,M,N,Z:integer;
    begin
      D:=D+1;
      write("⊞⇕⇕⇕⇓⇓⇓⇓");
      write(D:2);
      writeln();writeln();writeln();{.........3 CARRIAGE RETURNS }
      for N:=0 to 7 do
        begin
          write("     ⇕");
          for M:=0 to 7 do
            begin
              if X[N]=M then
                begin
                  F:=M+1;
                  write("Q⇕");
                  music("+A0")
                end
              else write(" ⇕")
            end;
          writeln("⇕",F:2,"⇓")
        end
    end;
begin { ..................................EIGHT QUEENS MAIN }
    tempo(7);range(C,40);
    CLEAR;
    BOARD;
    ARYWRITE
end.
```

(3) Calendar Program (Application Program 3)

When this sample program is executed, a message appears which requests the operator to enter the year. The program displays the calendar for month of January of the specified year and steps. Pressing any key advances the month.

This program is the third section of the PASCAL Applications Tape.

```
{ Our Calendar }
var YEAR,MONTH,TOP,Y:integer;LCHR:char;
procedure PRINTCALENDAR(Y,M,T:integer);{ .......... Print Calendar }
  var N,I,DAYS:integer;
  begin
    case M of 1,3,5,7,8,10,12:DAYS:=31;
                      4,6,9,11:DAYS:=30;
                        2:if(Y mod 4=0)and(Y mod 100<>0)or(Y mod 400=0)
                            then DAYS:=29
                            else DAYS:=28
    end;
    write("@");
    cursor(11,3);writeln("♦♦  ",Y:4," - ",M:2,"  ♦♦⇩");
    PRINTLINE('▬');
    write("    SUN  MON  TUE  WED  THU  FRI  SAT");
    PRINTLINE('—');
    writeln("⇩");
    for I:=1 to T do write("     ");
    repeat
      if(I<>1)and(I mod 7=1)then
        writeln("⇩");
      write(I-T:5);
      I:=I+1;
    until I-T>DAYS;
    TOP:=(I-1)mod 7;
    PRINTLINE('▬');
    music("+G0");
  end;
procedure PRINTLINE(LCHR:char);{ ....Carriage Return & Line Print }
var N:integer;
  begin
    writeln();
    for N:=1 to 39 do
      write(LCHR:1)
  end;
begin { ....................................... Main }
  range(C,40);tempo(6);
  while key=chr(0)do
    begin
      cursor(13,22);write("Year ");
      read(YEAR);
      Y:=YEAR-1;
      TOP:=(Y+(Y div 4)-(Y div 100)+(Y div 400)+1)mod 7;
      for MONTH:=1 to 12 do
        begin
          PRINTCALENDAR(YEAR,MONTH,TOP);
          while key=chr(0)do
        end
    end
end.
```

(4) Magic Square (Application Program 4)

A square grid is specified and numbers are assigned to all squares of the grid to that the total of the numbers on any horizontal vertical line or diagonal line are the same. The number of squares on one side of the grid must be an odd number from 3 to 19. When 9 or greater is specified, the result is output to the printer; otherwise, it is displayed on the CRT screen.

This program is the fourth section of the PASCAL Applications Tape.

```
{ A Mathematical Game : Magic Square }
var DATA:array[18,18]of integer;
    AREAD,XMAX,DATAN,Q,X,Y:integer;
    CH:char;
procedure ARRAYCLEAR;{ ............................Clears Array }
  var I,J:integer;
  begin
    for I:=0 to 18 do
      for J:=0 to 18 do DATA[I,J]:=0
  end;
procedure KEYIN;{ Displays Title and Reads Number of Squares
  var N:integer;                                     on a Side }
  begin
    repeat
      writeln("⊡⇩⇩⇩⇨⇨⇨⇨⇨⇨⇨◆◆◆ MATHEMATICAL GAME ◆◆◆");
      write("          ");
      for N:=0 to 24 do write("—");
      writeln();
      writeln("⇩⇩   Number of squares must be an odd⇩");
      writeln(" number [3-19].");
      writeln("⇩⇩⇩   When it is more than or equal to 9,⇩");
      writeln(" data is output to the Printer.");
      write("⇩⇩⇩⇩   Enter number of squares ");
      read(XMAX);
    until odd(XMAX)and(XMAX>2)and(XMAX<20)
  end;
procedure WBEGIN;
  begin
    X:=(XMAX-1)div 2;
    Y:=XMAX-1;
    DATAN:=1;
    ARRAYWRITE(DATAN,X,Y);
    DATAN:=DATAN+1;
    X:=X+1;
    Y:=0;
    ARRAYWRITE(DATAN,X,Y)
  end;
procedure ARRAYWRITE(N,XN,YN:integer);
  begin
    DATA[XN,YN]:=N
  end;
procedure DATAWRITE;
  var MAXSIZE:integer;
  begin
    MAXSIZE:=XMAX*XMAX;
    repeat
      DATAN:=DATAN+1;
      X:=X+1;
      Y:=Y+1;
      JUDGE
    until DATAN=MAXSIZE
```

```
  end;
procedure JUDGE;{ ............................ Check Data Area }
  var GMAX:integer;
  begin
    GMAX:=XMAX-1;
    if(X<XMAX)and(Y<XMAX)then
      begin
        ARRAYREAD(X,Y);
        if AREAD=0 then ARRAYWRITE(DATAN,X,Y)
          else begin
                 X:=X-1;
                 Y:=Y-2;
                 ARRAYWRITE(DATAN,X,Y)
               end
      end
    else if(X>GMAX)and(Y<XMAX)then
             begin
               X:=0;
               ARRAYWRITE(DATAN,X,Y)
             end
           else if(X<XMAX)and(Y>GMAX)then
                    begin
                      Y:=0;
                      ARRAYWRITE(DATAN,X,Y)
                    end
                  else if(X>GMAX)and(Y>GMAX)then
                         begin
                           X:=X-1;
                           Y:=Y-2;
                           ARRAYWRITE(DATAN,X,Y)
                         end
    end;
procedure ARRAYREAD(X,Y:integer);
  begin
    AREAD:=DATA[X,Y]
  end;
procedure DATAOUT;{ ......................Outputs Data to CRT }
  var M,N:integer;
  begin
    writeln("©  ** MATHEMATICAL GAME DATA **");
    if XMAX>8 then
      begin
        writeln("↓↓↓↓↓↓    Since data is too large,↓");
        writeln("  result is output to the Printer !↓");
        PRINTER;
      end
    else begin DATAPRINT;BOARD end
  end;
procedure DATAPRINT;
  var M,N:integer;
  begin
    Y:=XMAX;
    Q:=0;
    write("⊠↓↓↓");
    for M:=1 to XMAX do
      begin
        Y:=Y-1;
        X:=0;
        write(" ");
        for N:=1 to XMAX do
```

```
          begin
            ARRAYREAD(X,Y);
            write(AREAD:4);
            X:=X+1
          end;
        writeln("⇩⇩");
        Q:=AREAD+Q
      end
  end;
procedure BOARD;{..............................Write Board }
  var M,N:integer;
  begin
    M:=XMAX-1;
    UP;
    for N:=1 to M do begin
                       SIDE;
                       MID
                     end;
    SIDE;
    BOTTOM
  end;
procedure UP;
  var M,N:integer;
  begin
    M:=XMAX-1;
    write("▣⇩ ┌");
    for N:=1 to M do write("───┬");
    writeln("───┐");
  end;
procedure SIDE;
  var N,S:integer;
  begin
    for N:=1 to 2 do
      begin
        write(" │");
        for S:=1 to XMAX do write("⇔⇔⇔│");
        writeln();
      end
  end;
procedure MID;
  var M,N:integer;
  begin
    write(" ├");
    M:=XMAX-1;
    for N:=1 to M do write("───┼");
    write("───┤");
    writeln()
  end;
procedure BOTTOM;
  var M,N:integer;
  begin
    write(" └");
    M:=XMAX-1;
    for N:=1 to M do write("───┴");
    writeln("───┘");write("   Result");
    write(Q:5)
  end;
procedure PRINTER;{ ............................ Printer }
  var M,N,P,Q,R,S,T,U:integer;
  begin
```

```
    M:=XMAX-1;
    Y:=XMAX;
    Q:=0;
    X:=0;
    pwriteln();
    pwriteln(chr(14):1,"          ◆◆  MATHEMATICAL GAME  ◆◆",chr(20));
    pwriteln();
    pwrite(chr(27):1,chr(0):1,chr(9):1," ┌");
    for N:=1 to M do pwrite("───┬");
    pwriteln("───┐");
    for R:=1 to XMAX do
      begin
        pwrite(" │");
        for T:=1 to XMAX do pwrite("   │");
        pwriteln();
        Y:=Y-1;
        pwrite(" │");
        for P:=1 to XMAX do
          begin
            ARRAYREAD(X,Y);
            pwrite(AREAD:3,"│");
            X:=X+1
          end;
        X:=0;
        pwriteln();
        Q:=AREAD+Q;
        if R<XMAX then
          begin
            pwrite(" ├");
            for U:=1 to XMAX-1 do pwrite("───┼");
            pwriteln("───┤")
          end
      end;
    pwrite(" └");
    for S:=1 to M do pwrite("───┴");
    pwriteln("───┘",chr(10));
    pwriteln(chr(10));
    pwrite(chr(14):1,"  TOTAL OF NUMBERS :::::");
    pwriteln(Q:5,chr(20):1,chr(27):1,chr(2):1,chr(12):1);
  end;
begin { ....................................Main }
  range(C,40);
  repeat
    ARRAYCLEAR;
    KEYIN;
    WBEGIN;
    DATAWRITE;
    DATAOUT;
    writeln();
    write("   Continue or not (Y OR N) ");
    read(CH)
  until CH<>'Y'
end.
```

** MATHEMATICAL GAME **

192	213	234	255	276	297	318	339	360	1	22	43	64	85	106	127	148	169	190
212	233	254	275	296	317	338	359	19	21	42	63	84	105	126	147	168	189	191
232	253	274	295	316	337	358	18	20	41	62	83	104	125	146	167	188	209	211
252	273	294	315	336	357	17	38	40	61	82	103	124	145	166	187	208	210	231
272	293	314	335	356	16	37	39	60	81	102	123	144	165	186	207	228	230	251
292	313	334	355	15	36	57	59	80	101	122	143	164	185	206	227	229	250	271
312	333	354	14	35	56	58	79	100	121	142	163	184	205	226	247	249	270	291
332	353	13	34	55	76	78	99	120	141	162	183	204	225	246	248	269	290	311
352	12	33	54	75	77	98	119	140	161	182	203	224	245	266	268	289	310	331
11	32	53	74	95	97	118	139	160	181	202	223	244	265	267	288	309	330	351
31	52	73	94	96	117	138	159	180	201	222	243	264	285	287	308	329	350	10
51	72	93	114	116	137	158	179	200	221	242	263	284	286	307	328	349	9	30
71	92	113	115	136	157	178	199	220	241	262	283	304	306	327	348	8	29	50
91	112	133	135	156	177	198	219	240	261	282	303	305	326	347	7	28	49	70
111	132	134	155	176	197	218	239	260	281	302	323	325	346	6	27	48	69	90
131	152	154	175	196	217	238	259	280	301	322	324	345	5	26	47	68	89	110
151	153	174	195	216	237	258	279	300	321	342	344	4	25	46	67	88	109	130
171	173	194	215	236	257	278	299	320	341	343	3	24	45	66	87	108	129	150
172	193	214	235	256	277	298	319	340	361	2	23	44	65	86	107	128	149	170

TOTAL OF NUMBERS ::::: 3439

(5) Hexadecimal-to-decimal Conversion Program

The following sample program performs the step described on page 81.

```
{ Hexadecimal-to-decimal Conversion }
var DATA:array[10]of char;
    A:integer;
    FLUG:boolean;
procedure DATAIN;
  var N:integer;
      X:char;
  begin write("Enter data in hex. $");
    for N:=0 to 10 do DATA[N]:='0';
    N:=0;X:='X';
    while X<>chr(13)do { ............................13 is CR }
      begin
        repeat
          X:=key
        until((X>'/')and(X<':'))or((X>'@')and(X<'G'))or(X=chr(13));
        if X<>chr(13)then write(X:1);
        DATA[N]:=X;
        N:=N+1
      end;
    if N>5 then begin
                  writeln("  Input Error ⇩");
                  DATAIN
                end
    else begin
           case N of 4:begin
                         DATA[4]:=DATA[3];
                         DATA[3]:=DATA[2];
                         DATA[2]:=DATA[1];
                         DATA[1]:=DATA[0];
                         DATA[0]:='0'
                       end;
                     3:begin
                         DATA[4]:=DATA[2];
                         DATA[3]:=DATA[1];
                         DATA[2]:=DATA[0];
                         DATA[1]:='0';
                         DATA[0]:='0'
                       end;
                     2:begin
                         DATA[4]:=DATA[1];
                         DATA[3]:=DATA[0];
                         DATA[2]:='0';
                         DATA[1]:='0';
                         DATA[0]:='0'
                       end;
                     1:for N:=0 to 3 do DATA[N]:='0';
           end
         end
  end;
procedure TRANS;{ ...............Conversion of less than $8000 }
  begin
    if DATA[0]<'8'then
      begin A:=(ord(DATA[0])-48)*4096;
        if DATA[1]>'9'then A:=A+(ord(DATA[1])-55)*256
```

```
     else A:=A+(ord(DATA[1])-48)*256;
    if DATA[2]>'9'then A:=A+(ord(DATA[2])-55)*16
     else A:=A+(ord(DATA[2])-48)*16;
    if DATA[3]>'9'then A:=A+(ord(DATA[3])-55)
     else A:=A+(ord(DATA[3])-48);
     writeln(" =",A:6,"⇩");
    if A=0 then FLUG:=false
   end
  else TRANS1
 end;
procedure TRANS1;{ ............ Conversion of greater than $7FFF }
 var B:real;
 begin
  if(DATA[0]='8')and(DATA[1]='0')and(DATA[2]='0')and(DATA[3]='0')
     then writeln(" =  ` [-32767-1]⇩")
   else begin
       if DATA[0]>'9'then B:=float(ord(DATA[0])-55)*4096.0
        else B:=float(ord(DATA[0])-48)*4096.0;
       if DATA[1]>'9'then B:=B+float(ord(DATA[1])-55)*256.0
        else B:=B+float(ord(DATA[1])-48)*256.0;
       if DATA[2]>'9'then B:=B+float(ord(DATA[2])-55)*16.0
        else B:=B+float(ord(DATA[2])-48)*16.0;
       if DATA[3]>'9'then B:=B+float(ord(DATA[3])-55)
        else B:=B+float(ord(DATA[3])-48);
       A:=trunc(65536.0-B)*(-1);
       writeln(" =",A:10);
       writeln()
      end
 end;
begin { ............................................... MAIN }
 writeln("©◆◆Hexadecimal-to decimal Conversion⇩⇩");
 FLUG:=true;
 repeat
  DATAIN;
  TRANS
 until not FLUG
end.
```

(6) Conversion Program for decimal numbers in the range from –32767 to +32767 into hexadecimals

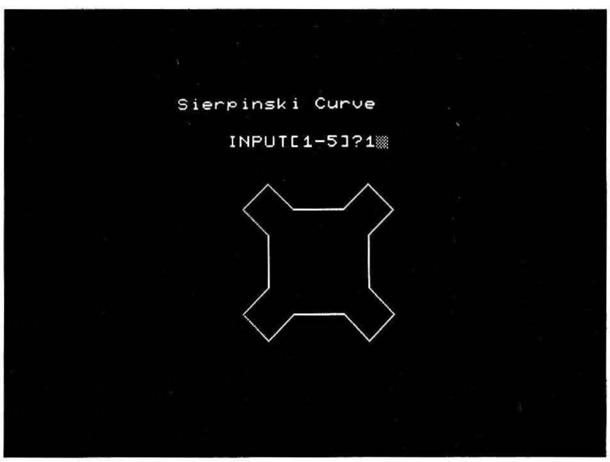
```
{ Decimal-to Hexadecimal Conversion }
var A,DATA:integer;
    DATA1:real;
    FLUG:boolean;
procedure DATAIN;{ ...............................Read Data }
  begin
    write("Enter Data in Decimal [+32767....-32767] ");
    readln(DATA);
    DATA1:=float(DATA);
    if DATA=0 then FLUG:=false
      else begin
             FLUG:=true;
             if DATA<0 then TRANS16R(DATA1)
               else begin A:=DATA div 4096;TRANS16 end
           end
  end;
procedure TRANS16;{.................... Positive Data Processing }
  begin
    write("$ ");
    if A<>0 then begin DISP1(A);DATA:=DATA mod 4096 end
      else write("0");
    A:=DATA div 256;
    if A<>0 then begin DISP1(A);DATA:=DATA mod 256 end
      else write("0");
    A:=DATA div 16;
    if A<>0 then begin DISP1(A);DATA:=DATA mod 16 end
      else write("0");
    DISP1(DATA);
    writeln()
  end;
procedure TRANS16R(NEG:real);{ .........Negative Data Processing }
  var X,Y:real;
  begin
    X:=65536.0+NEG;
    A:=trunc(X/4096.0);
    Y:=float(A);
    DATA:=trunc(X-Y*4096.0);
    TRANS16
  end;
procedure DISP1(Z:integer);
  begin
    if Z>9 then case Z of 10:write("A");
                          11:write("B");
                          12:write("C");
                          13:write("D");
                          14:write("E");
                          15:write("F")
                     end
     else write(Z:1)
  end;
begin { ....................................... Main }
  write("@");
  FLUG:=true;
  repeat
    DATAIN
  until not FLUG
end.
```

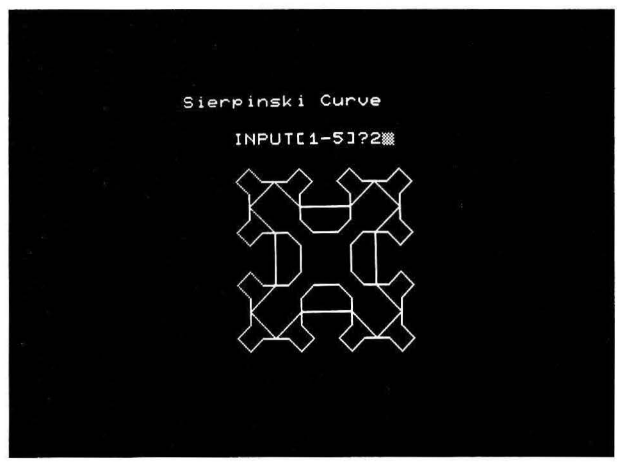
(7) Sierpinski Curve

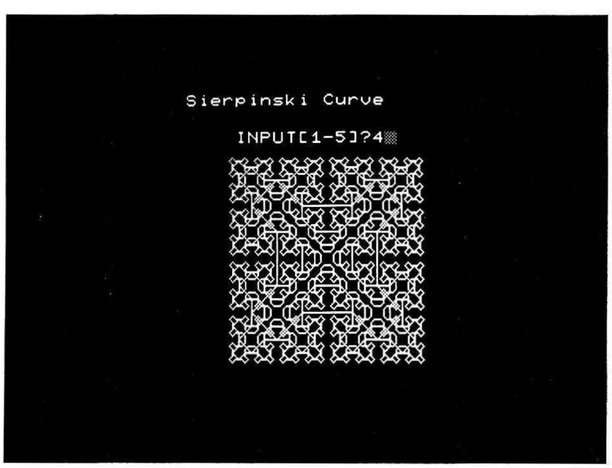
This sample program controls the graphic display control. Therefore, graphic RAM expansion is required. The number of size levels of Sierpinski curves is from 1 through 5.

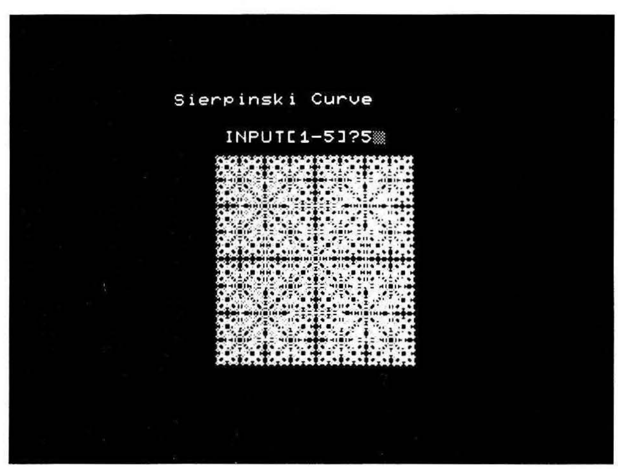
```
{ Sierpinski Curve }
var P,X,Y,X1,Y1,H,I,N:integer;
      R:char;
{ .................................... Sierpinski }
procedure AA(I:integer);
  begin if I>0 then
          begin AA(I-1);X:=X+H;Y:=Y-H;PLOT;
                BB(I-1);X:=X+H+H;PLOT;
                DD(I-1);X:=X+H;Y:=Y+H;PLOT;
                AA(I-1)
          end
  end;
procedure BB(I:integer);
  begin if I>0 then
          begin BB(I-1);X:=X-H;Y:=Y-H;PLOT;
                CC(I-1);Y:=Y-H-H;PLOT;
                AA(I-1);X:=X+H;Y:=Y-H;PLOT;
                BB(I-1)
          end
  end;
procedure CC(I:integer);
  begin if I>0 then
          begin CC(I-1);X:=X-H;Y:=Y+H;PLOT;
                DD(I-1);X:=X-H-H;PLOT;
                BB(I-1);X:=X-H;Y:=Y-H;PLOT;
                CC(I-1)
          end
  end;
procedure DD(I:integer);
  begin if I>0 then
          begin DD(I-1);X:=X+H;Y:=Y+H;PLOT;
                AA(I-1);Y:=Y+H+H;PLOT;
                CC(I-1);X:=X-H;Y:=Y+H;PLOT;
                DD(I-1)
          end
  end;
procedure PLOT;{ ................. Draw Line between Two Points }
  begin
    line(X1,Y1,X,Y);
    X1:=X;Y1:=Y
  end;
procedure HOW;{ ............................ Read Size Number }
  begin
    repeat
      writeln("@→→→→→→→→ Sierpinski Curve ");
      write("↓↓                INPUT[1-5]");
      readln(R);
    until(R>'0')and(R<'6');
    N:=ord(R)-48
  end;
{ ....................................Sierpinski Main }
begin
  range(C,40);
  repeat
```

```
    HOW;
    graph(I,1,C,0,1);
    I:=0;H:=32;
    X:=2*H+90;
    Y:=3*H+38;
    repeat
      I:=I+1;
      X:=X-H;
      H:=H div 2;
      Y:=Y+H;
       X1:=X;Y1:=Y;
      AA(I);X:=X+H;Y:=Y-H;PLOT;
      BB(I);X:=X-H;Y:=Y-H;PLOT;
      CC(I);X:=X-H;Y:=Y+H;PLOT;
      DD(I);X:=X+H;Y:=Y+H;PLOT;
    until I=N
  until key='E'
end.
```







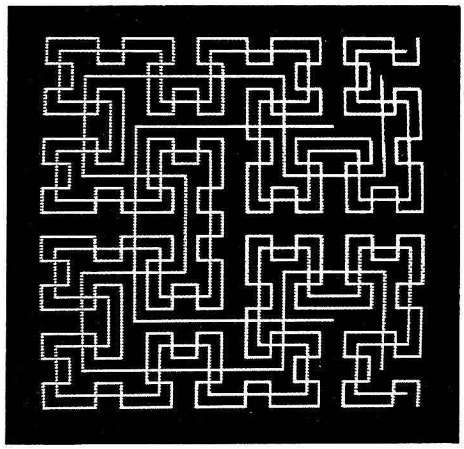


(8) Color Hilbert Curve

This sample program is an example of color system control program. The color control program (SB-3000 series) must be loaded in memory in advance.

The number of size levels of the curve is from 1 through 7.

```
{ Color .................. HILBERT }
var P,X,Y,X1,Y1,H,I,N,X0,Y0:integer;
    R:char;
procedure TRANSL(A,B,C,D:integer);
  begin
    tran('L',',');
    TRANS(A,B);tran(',');
    TRANS(C,D);tran(chr(13))
  end;
procedure TRANS(A,B:integer);
  begin
    COLOR(A);tran(',');
    COLOR(B)
  end;
procedure COLOR(X:integer);{ Convert X into ASCII Code , and
  var C100,C10:char;                                        Transfer }
  begin
    C100:=chr(X div 100+48);
    X:=X mod 100;
    C10:=chr(X div 10+48);
    tran(C100,C10,chr(X mod 10+48))
  end;
{ ..........................................HILBERT }
procedure AA(I:integer);
  begin if I>0 then
          begin DD(I-1);X:=X-H;PLOT;
                AA(I-1);Y:=Y-H;PLOT;
                AA(I-1);X:=X+H;PLOT;
                BB(I-1)
          end
  end;
procedure BB(I:integer);
  begin if I>0 then
          begin CC(I-1);Y:=Y+H;PLOT;
                BB(I-1);X:=X+H;PLOT;
                BB(I-1);Y:=Y-H;PLOT;
                AA(I-1)
          end
  end;
procedure CC(I:integer);
  begin if I>0 then
          begin BB(I-1);X:=X+H;PLOT;
                CC(I-1);Y:=Y+H;PLOT;
                CC(I-1);X:=X-H;PLOT;
                DD(I-1)
          end
  end;
procedure DD(I:integer);
  begin if I>0 then
          begin AA(I-1);Y:=Y-H;PLOT;
                DD(I-1);X:=X-H;PLOT;
                DD(I-1);Y:=Y+H;PLOT;
                CC(I-1)
          end
```



```
  end;
procedure PLOT;{ .................Draw Line between Tow Points }
  begin
    TRANSL(X,Y,X1,Y1);
    X1:=X;Y1:=Y
  end;
procedure HOW;{ ..............................Read Size Number }
  begin
    repeat
      writeln("@⇩⇩⇩⇩⇩⇨⇨⇨⇨⇨⇨⇨Color HILBERT Curve ");
      write("⇩⇩⇩⇨⇨Enter size Number [ 1--7 ] ");
      readln(R);
      write("⇩⇩⇩⇩⇨⇨⇨⇨⇨⇨⇨⇨⇨◆◆◆  Running  ◆◆◆");
    until(R>'0')and(R<'8');
    N:=ord(R)-48
  end;
{ ................................................HILBERT Main }
begin
  repeat
    HOW;
    tran("M,0","B,0");{ ........8 Color Mode, Black Background }
    for P:=-1 to 6 do
      begin
        I:=0;
        H:=128;
        X0:=H div 2+64;
        Y0:=X0-35;
        repeat
          I:=I+1;
          H:=H div 2;
          Y0:=Y0+H div 2;
          X0:=X0+H div 2;
          X:=X0;
          Y:=Y0;
          X1:=X;
          Y1:=Y;
          tran('C',',',chr((P+I)mod 7+49),chr(13));
          AA(I)
        until I=N
      end
  until key='E'
end.
```

(9) Port I/O Program

This sample program transfers data between the computer and the color control terminal via the port by means of *input* and *output* statements.

Procedure COLOR (line 2 through 11) performs almost the same function as *tran* (A). Refer to the OUTPUT MODE routine in the Color Control Manual.

Key in M, 0 CR B, 2 CR C, 1 CR SF, 127, 95, 0, SHARP CR in succession; the result will be the same as that of example 1 on page 94.

```
 0.{ I/O Control Program via Port }
 1.var A,B,C:char;
 2.procedure COLOR(A:char);
 3.   var B:char;
 4.   begin
 5.     B:=chr(1);
 6.     repeat
 7.     until B=chr(ord(input(239))mod 2);{ ...........Check Bit 0 }
 8.     output(A,238);{ ............Output Key in Data to Port $EE }
 9.     output(chr(6),239);{ .................Output 6 to Port $EF }
10.     output(chr(7),239){ ..................Output 7 to Port $EF }
11.   end;
12.begin
13.   B:='!';{ ..............................Dummy for Repetition }
14.   write("@↓↓↓?");
15.   repeat
16.     repeat
17.       A:=key;
18.       if A=chr(102)then A:=chr(13)
19.     until A<>chr(0);
20.     COLOR(A);
21.     if A=chr(13)then begin writeln();write("?")end
22.       else write(A:1)
23.   until B=key
24.end.
25.
```

SHARP CORPORATION

MODEL: MZ8BT02E

TINSE0031PAZZ
080261-010881

MZ-80B
E1