Chapter 4

Programming the CBM

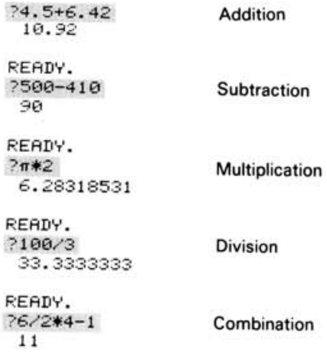
This chapter teaches you how to start writing your own BASIC programs.

BASIC is a programming language. BASIC, like any programming language, consists of a set of statements, which you combine to create programs. A program defines the task you want the computer to perform.

We could teach you BASIC by forcing you first to learn BASIC statements, one by one. But you would probably give up, since individual statements are not very meaningful. A study of individual BASIC statements quickly degenerates into learning a number of arbitrary syntax rules that tell you nothing about programming or good programming practice. Therefore **rigorous definitions of all BASIC statements have been relegated to Chapter 8. Look up individual statements in Chapter 8 when you need to,** but do not try to read Chapter 8 before you read this chapter.

IMMEDIATE AND PROGRAMMED MODES

When the CBM computer is powered up it is in immediate mode. In immediate mode you can use the CBM computer as you would a calculator; it executes BASIC statements as soon as you press the RETURN key to signal the end of the statement entry. Try these arithmetic examples:



```
?4.5+6.42            Addition
 10.92

READY.
?500-410             Subtraction
 90

READY.
?π*2                 Multiplication
 6.28318531

READY.
?100/3               Division
 33.3333333

READY.
?6/2*4-1             Combination
 11
```

Results are displayed immediately on the next line of the display.

In programmed mode the computer accepts and stores your entries, but does not perform any operations until specifically instructed to do so by a RUN statement.

Programs and Statements

Each of the five immediate mode statements shown above is a miniature program.

A program provides the CBM computer with an exact and complete definition of the task which the computer is to perform.

A program consists of one or more statements. In each of the five immediate mode illustrations, the entire program consists of a single statement. These are trivial cases. Most programs have tens, hundreds, or even thousands of statements.

Program Execution

A computer is said to execute a program (or RUN the program) when it performs the operations which the program specifies.

An immediate mode program is executed as soon as you press the RETURN key.

In programmed mode you must issue a special RUN statement to execute a program; we described the RUN statement in Chapter 1.

Program Lines

In programmed mode every program line has a unique line number. The CBM computer assumes an immediate mode program if the line does not begin with a line number.

A program line can be up to 80 characters long. On an 80-column display, therefore, a program line corresponds to a display line. On a 40-column display a program line is equivalent to two display lines.

If a program line is less than 80 characters long, then it is terminated when you press the RETURN key. The CBM computer lets you continue beyond the 80th character, but subsequently the line does not execute correctly. To be safe you should end every line before the 80th character by pressing the RETURN key.

A line can contain more than one program statement, providing, of course, the entire line length is less than 80 characters. This holds true in program mode and in immediate mode.

ONE-LINE IMMEDIATE MODE PROGRAMS

In immediate mode the entire program must fit on a single line, since the immediate mode program is executed as soon as you press the RETURN key. A single line can contain more than one statement, therefore some interesting immediate mode programs can be created. Let us examine some possibilities.

A question mark appearing at the beginning of a BASIC statement causes the CBM computer to display something; the question mark is an abbreviated form of the PRINT statement. Although the illustrations of immediate mode statements shown earlier all begin with a question mark, this is by no means a requirement for an immediate mode program. Consider the following examples:

```
A=π*2

READY.
?A
 6.28318531
```

There are two immediate mode statements. Each becomes an independent, immediate mode program. When you type in the first statement, A=$\pi \cdot 2$, the result is not displayed, since the statement does not begin with ?; but the calculation is performed nevertheless. The result is displayed by the second immediate statement, ?A.

When statements are grouped together on one line, each is separated from the next with a colon (:). Thus, the two statements:

```
A=π*2
?A
```

can be condensed into one line as follows:

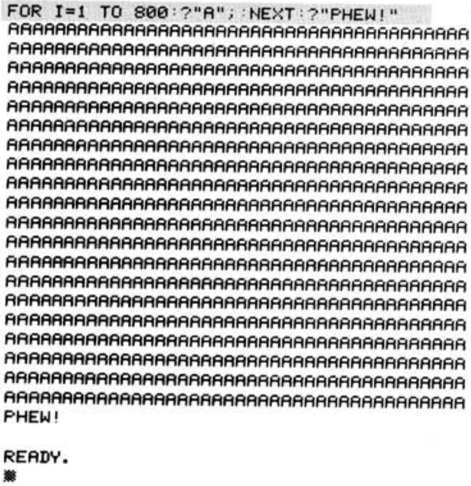
```
A=π*2:?A
```

The two statements have become a single, immediate mode program.

Since a line can have up to 80 characters, you can put a lot of program on one line, and execute it all in immediate mode. For example, consider the following line:

```
FOR I=1 TO 800:?"A";:NEXT:?"PHEW!"
```

Ignoring the meaning of this "mini-program" for now, type it in exactly as shown, ending with a RETURN. If you type it in successfully, you will see the letter A displayed across the next 20 lines of a 40-column screen, followed by the message PHEW! on the 21st line:



```
FOR I=1 TO 800:?"A";:NEXT:?"PHEW!"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
PHEW!

READY.
▒
```

The program line is conveniently left at the top of the screen. This is because the program displays just enough lines to scroll the program line to the top of a 40-column screen, but not off it.

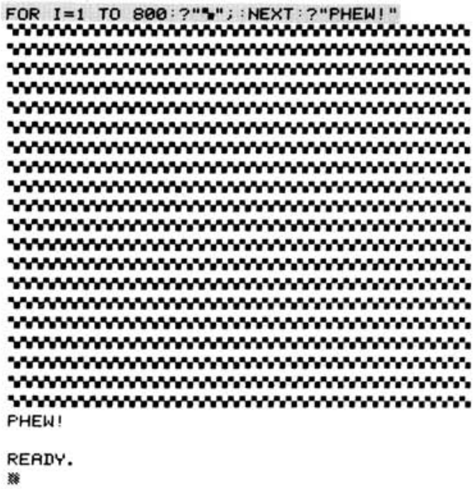
The letter A will be displayed across 10 lines of an 80-column screen, with the program above the top line of A's.

Re-executing in Immediate Mode

When the one-line program described above completes execution in immediate mode, the READY message is displayed and the cursor is left at the beginning of the bottom display line.

An important feature of CBM BASIC is that **anything displayed on the screen is "live." You can edit any line on the screen and re-execute the edited statements,** providing they are still displayed.

Use CURSOR UP or, more conveniently, press the HOME key to move the cursor up to the F in FOR. Move the cursor right 15 positions to the A. Press a graphic key, say the DIAGONAL QUARTER-BLOCK SOLID (shift of ? key). Press RETURN. The new symbol now overwrites and replaces all the A's across the 20-row display. On completion, the cursor again rests at the beginning of the bottom line.



```
FOR I=1 TO 800:?"▚";:NEXT:?"PHEW!"
PHEW!

READY.
```

Modifying a Program

Before trying any more characters, make one editing modification to the line to make changing characters easier. The new line, with the display character changed to a W, will look like this:

```
C$="W":FOR I=1 TO 800:?C$;:NEXT:?"PHEW!"
```

To modify the current line, perform the following steps:

1. Home the cursor so it is blinking at the F in FOR (▒ indicates position of cursor).

```
FOR I=1 TO 800:?"▚";:NEXT:?"PHEW!"
```

2. Press the INSERT key seven times.

```
        FOR I=1 TO 800:?"▚";:NEXT:?"PHEW!"
```

3. Type in the seven characters *C$="W":*

```
C$="W":FOR I=1 TO 800:?"▚";:NEXT:?"PHEW!"
```

4. CURSOR RIGHT 14 times to the first quotation mark.

```
C$="W":FOR I=1 TO 800:?"▚";:NEXT:?"PHEW!"
```

5. Type in the two characters *C$*

```
C$="W":FOR I=1 TO 800:?C$";:NEXT:?"PHEW!"
```

6. Remove the other quotation mark by pressing one CURSOR RIGHT:

```
C$="W":FOR I=1 TO 800:?C$";:NEXT:?"PHEW!"
```

Followed by one DELETE:

```
C$="W":FOR I=1 TO 800:?C$;:NEXT:?"PHEW!"
```

The changes have all been made; press RETURN to print the new character. Now you can HOME the cursor, then move it right just four positions to change the display character. Display any other characters you want. The graphics are especially interesting.

SPACES ARE NOT NEEDED

Are you struggling with the question of where to put spaces in the line and where not to? Don't worry. **CBM BASIC interprets a line by the elements in it. Spaces, or blanks, are irrelevant.** For example, the line:

```
120 FOR I=1 TO 210
```

could read:

```
120 FOR I=1 TO210
```

or:

```
120 FORI=1TO210
```

You can put extra spaces anywhere, except within reserved words or other BASIC statements. GOTO may be written as either GOTO or GO TO. The only place you must put spaces is within quotation marks, where you want spaces to be part of the text string. Blanks in a statement improve readability of the program; use them for this purpose.

ELEMENTS OF A PROGRAMMING LANGUAGE

Program statements must be written following a well defined set of rules. These rules, taken together, are referred to as "syntax."

There are many different sets of rules, or syntax, that define the way in which program statements are written. Each different set of rules applies to a different programming language. **CBM computers use** just one programming language; it is called **BASIC.** All of the syntax rules described in this book apply only to CBM BASIC.

Programming languages are as varied as spoken languages. In addition to BASIC, other common programming languages are PASCAL, FORTRAN, COBOL, APL, PL/M, PL-1, and FORTH. Uncommon program languages number in the hundreds.

Unfortunately, programming languages, like spoken languages, have dialects. **A BASIC program written for your CBM computer will not run on any other computer,** even if the other computer also claims to be programmable in BASIC. Dialects manifest themselves as minor variations in the language syntax used by one computer as compared to another. However, having learned how to program your CBM computer in BASIC, you will have little trouble learning any other computer's BASIC.

Some programming language syntax rules are obvious. The addition and subtraction examples at the beginning of this chapter use obvious syntax. You do not have to be a programmer to understand these two statements. But most syntax rules are utterly arbitrary; they are meaningless unless you have learned the syntax. You should not try to seek justification for syntax rules; usually there is none. For example, why use "*" to represent multiplication? One would normally use a "×" sign for multiplication; but the computer would have no way of differentiating between the use of the "×" sign to represent multiplication, or to represent the letter "x". Therefore nearly all computer languages have opted for the asterisk (*) to represent multiplication. Division is universally represented by the "/" sign. There is no real justification for this selection; the standard division sign (÷) is not present on computer or typewriter keyboards, so some other character must be selected.

BASIC statement syntax deals separately with line numbers, data, and instructions to the computer. We will describe each in turn.

LINE NUMBERS

As we have already stated, in program mode **every line of a BASIC program must have a unique line number.** Moreover, the first line of the BASIC program must have the smallest line number, while the last line of the BASIC program must have the largest line number. In between, line numbers must be in ascending order. The CBM computer forces this upon you: **irrespective of where you enter a line on the display, the CBM computer will move it to its proper sequential position.** Consider an existing program with the following line numbers:

120
130
140
150
160
170
180
190

If you enter a new statement with line number 165, then the new statement initially appears below the existing program, but the CBM computer will automatically insert this statement between line numbers 160 and 170. This may be illustrated as follows:

Displayed line numbers when you entered line 165	Lines stored and re-displayed thus
120	120
130	130
140	140
150	150
160	160
170	165
180	170
190	180
	190
165	

If the line number for a new statement duplicates an existing line number, then the old statement will be replaced.

CBM BASIC allows line numbers to range between 1 and 63999. The CBM computer interprets digits appearing at the beginning of any line as the line number. If more than five digits appear at the beginning of the line then an error is flagged: it is referred to as a syntax error, since you have violated the syntax rules for CBM BASIC.

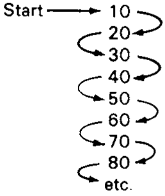
All BASIC dialects require line numbers to be assigned in ascending order as described above. However, the largest allowed line number varies from one dialect of BASIC to the next.

Computer languages other than BASIC do not require every line to begin with a line number, nor do they require line numbers, where present, to have any particular order.

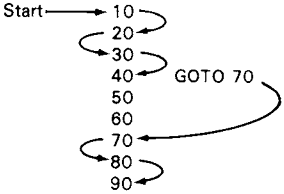
You use line numbers as addresses, identifying locations within a program. This is an important concept, since every program will contain two types of statements:

1. Statements which create or modify data, and
2. Statements which control the sequence in which operations are performed.

The idea that operations specified by a program must be performed in some well defined sequence is a simple enough concept. Normally program execution begins with the first statement in the program, and continues sequentially. This may be illustrated as follows:



But we will soon discover that most programs contain some non-sequential execution sequences. That is when line numbers become important, because you use the line number to identify a change in execution sequence. This may be illustrated as follows:



DATA

The statement (or statements) following a line number specify operations that the computer is to perform, as well as data that must be used while performing these operations. We will now describe the types of data you may encounter in a CBM BASIC program.

There are two kinds of numbers that can be stored in CBM computers: floating point numbers (also called real numbers) and integers.

Floating Point Numbers

Floating point is the standard number representation used by CBM computers. All arithmetic is done using floating point numbers. **A floating point number can be a whole number, or a fractional number preceded by a decimal point.** The number can be negative (−) or positive (+). If the number has no sign it is assumed to be positive. Here are some examples of floating point numbers that are equivalent to integers:

```
5
-15
65000
161
0
```

Here are examples of floating point numbers that include a decimal point:

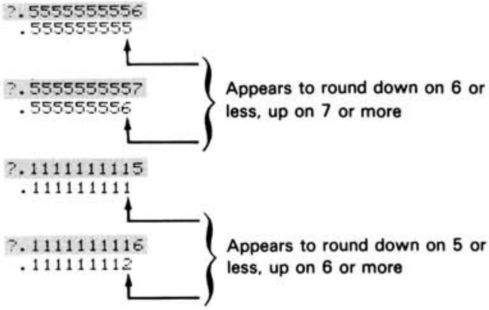
```
0.5
0.0165432
-0.0000009
1.6
24.0055
-64.2
3.1416
```

Note that if you put commas in a number, you will get a SYNTAX ERROR message. For example, use 65000, not 65,000.

Roundoff

Numbers always have at least eight digits of precision; they can have up to nine, depending on the number. CBM BASIC rounds off additional significant digits. Usually it rounds up when the next digit is five or more, and it rounds down when the next digit is four or less, but there are some roundoff quirks.

Here are some examples:



```
?.5555555556
 .555555555

?.5555555557
 .555555556

?.1111111115
 .111111111

?.1111111116
 .111111112
```

Appears to round down on 6 or less, up on 7 or more

Appears to round down on 5 or less, up on 6 or more

Scientific Notation

Large floating point numbers are represented using scientific notation. **CBM BASIC automatically converts numbers less than .01 or greater than 10^8 in magnitude to scientific notation.** Here are some examples:

```
READY.
?1111111114
1.11111111E+09

READY.
?1111111115
1.11111112E+09
```

A number in scientific notation has the form:

```
numberE+ee
```

where:

number	is an integer, fraction, or combination, as illustrated above. The "number" portion contains the number's significant digits; it is called the "coefficient." If no decimal point appears, it is assumed to be to the right of the coefficient.
E	is always the letter E. It substitutes for the word "exponent."
+	is an optional plus sign or minus sign.
ee	is a one-digit or two-digit exponent. The exponent specifies the magnitude of the number, that is, the number of places to the right (positive exponent) or to the left (negative exponent) that the decimal point must be moved to give the true decimal point location.

Here are some examples:

Scientific Notation	Standard Notation
2E1	20
10.5E+4	105000
66E+2	6600
66E−2	0.66
−66E−2	−0.66
1E−10	0.0000000001
94E20	9400000000000000000000

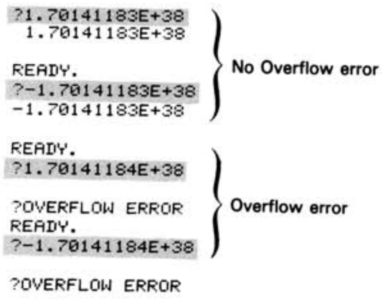
Scientific notation is a convenient way of expressing very large or very small numbers. CBM BASIC prints numbers ranging between 0.01 and 999,999,999 using standard notation; but numbers outside of this range are printed using scientific notation. Here are some examples:

```
?.009
 9E-03

READY.
?.01
 .01

READY.
?999999998.9
 999999999

READY.
?999999999.6
 1E+09
```

Even using scientific notation there is a limit to the size of a number that CBM BASIC can handle. The limits are:

Largest floating point number: +1.70141183E+38
Smallest floating point number: +2.93873588E−39

Any number of larger magnitude will give an overflow error. Here are some examples of overflow error:

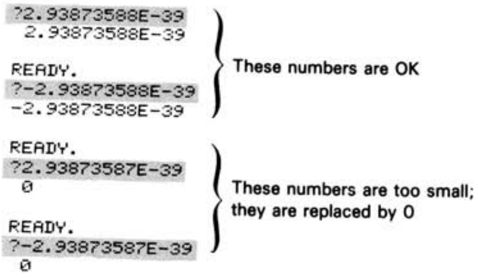


```
?1.70141183E+38
 1.70141183E+38

READY.
?-1.70141183E+38
-1.70141183E+38

READY.
?1.70141184E+38

?OVERFLOW ERROR
READY.
?-1.70141184E+38

?OVERFLOW ERROR
```

No Overflow error

Overflow error

A number that is smaller than the smallest magnitude will yield a zero result. This may be illustrated as follows:



```
?2.93873588E-39
 2.93873588E-39

READY.
?-2.93873588E-39
-2.93873588E-39

READY.
?2.93873587E-39
 0

READY.
?-2.93873587E-39
 0
```

These numbers are OK

These numbers are too small; they are replaced by 0

Integers

An integer is a number that has no fraction or decimal point. The number can be negative (−) or positive (+). An unsigned number is assumed to be positive. Integer numbers must have values in the range −32767 to +32768. The following are examples of integers:

0
1
44
32699
−15

Any integer can also be represented as a floating point number, since integers are a subset of floating point numbers. **CBM BASIC automatically converts integer numbers to floating point representation before using them in arithmetic.**

Strings

The word "string" is used to describe data that consists of words. This is non-numeric data; it is text.

We have already used strings as messages to be displayed on the CBM computer screen. **A string consists of one or more characters enclosed in double quotation marks.** Here are some examples of strings:

```
"HI!"
"SYNERGY"
"12345"
"$10.44 IS THE AMOUNT"
"22 UNION SQUARE, SAN FRANCISCO, CA"
```

Within a string you can include any alphabetic or numeric characters, special symbols or graphic characters, cursor control characters (CLEAR SCREEN/HOME, CURSOR UP/DOWN, CURSOR LEFT/RIGHT) and the REVERSE ON/OFF key. The only keys that cannot be used within a string are RUN/STOP, RETURN, and INSERT/DELETE.

All characters within the string are displayed as they appear. The cursor control and REVERSE ON/OFF keys, however, normally do not print anything themselves; to show that they are present in a string, certain reverse field symbols are used, as shown in Table 4-1.

Strings are entered as part of a statement. Since a statement must fit within an 80-character line, the longest string you can enter at a keyboard will have less than 80 characters; the statement needs some character positions for the line number, and required statement syntax.

Strings of up to 255 characters can be stored in CBM computer memory. Long strings are generated by concatenating shorter strings. We will describe how this is done later.

Variables

Earlier, when describing immediate mode, we illustrated the two-statement program:

```
A=π*2
?A
```

We rewrote the program using one statement:

```
A=π*2:?A
```

In these programs, A is a variable name.

The concept of a variable is easy to understand. Consider the two statements:

```
100 A=B+C
200 ?A
```

These two statements cause the sum of two numbers to be displayed. But what are the two numbers that get summed? They are whatever B and C represent at the time the statements are executed. In the following example:

```
90 B=4.65
95 C=3.72
100 A=B+C
200 ?A
```

B is assigned the value 4.65, while C is assigned the value 3.72. Therefore A equals 8.37.

Table 4-1. Special String Symbols

Function		Key	String Symbol*
Reverse On		OFF RVS ON	▟ (Reverse R)
Reverse Off	Shifted	OFF RVS ON	▀ (Reverse Shifted R)
Home Cursor		CLR SCREEN HOME	▙ (Reverse S)
Clear Screen	Shifted	CLR SCREEN HOME	▛ (Reverse Shifted S)
Cursor Down		CURSOR ↑↓	▖ (Reverse Q)
Cursor Up	Shifted	CURSOR ↑↓	▜ (Reverse Shifted Q)
Cursor Right		⇐ CURSOR ⇒	▐ (Reverse])
Cursor Left	Shifted	⇐ CURSOR ⇒	▌ (Reverse Shifted])

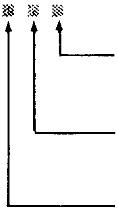
* The graphic symbol shown in this column may vary from one CBM computer to the next, depending on the computer's keyboard options. But the key description is accurate in every case.

Variable names can be used to represent string data or numeric data.

If you have studied elementary algebra, you will have no trouble understanding the concept of variables and variable names. If you have never studied algebra, then think of a variable name as a name which is assigned to a mail box. Anything which is placed in the mail box becomes the value associated with the mail box name.

Variable Names

A variable name can have one, two or three characters. The following character options are allowed:



Third character must be $ for a string variable, or % for an integer variable. A floating point variable name can only have two characters.

Second character can be any unshifted letter (A to Z) or any numeric digit (1, 2, 3, 4, 5, 6, 7, 8, 9, 0), for any type of variable.

First character must be an unshifted letter (A to Z) for any type of variable.

Thus the last character of the variable name tells CBM BASIC which type of data the variable represents.

Note that unshifted letters of the alphabet are used for the first and second label character. Depending on the model of CBM computer, the unshifted letter may be upper case or lower case. But in either case it is the letter displayed when the SHIFT key is not being depressed.

Floating point variables are the ones most frequently used in CBM BASIC. Here are some examples of floating point variable names:

```
A
B
C
A1
AA
Z5
```

Here are some examples of integer variable names:

```
A%
B%
C%
A1%
MN%
X4%
```

Remember, **floating point variables can have values that are equivalent to integers.** Here are examples of string variable names:

```
A$
M$
MN$
M1$
ZX$
F6$
```

Variable names can have more than two alphanumeric characters, but only the first two characters count. Therefore BANANA and BANDAGE are interpreted as the same name, since both begin with BA. CBM BASIC allows variable names to have up to 255 characters. Here are some examples of variable names with more than two characters:

MAGIC$	*interpreted as*	MA$
N123456789	*interpreted as*	N1
MMM$	*interpreted as*	MM$
ABCDEF%	*interpreted as*	AB%
CALENDAR	*interpreted as*	CA

If you use variable names with more than two characters, keep the following points in mind:

1. Only the first two characters, plus the identifier symbol ($ or %) are significant. Do not use extended names like LOOP1 and LOOP2; these are interpreted as the same variable: LO.
2. CBM BASIC has a number of "reserved words," which have special meaning within a BASIC statement. No variable name can contain a reserved word embedded anywhere in the name. Reserved words are listed in Table 4-4.
3. Additional characters need extra memory space, which you might need for longer programs. But the advantage of using longer variable names is that they make programs easier to read. PARTNO, for example, is more meaningful than PA as a variable name describing part numbers in an inventory program.

OPERATORS

The BASIC statement:

```
100 ?10.2+4.7
```

tells the CBM computer to add 10.2 and 4.7, and then display the sum. The statement:

```
250 C=A+B
```

tells the CBM computer to add the two floating point numbers represented by the variable names A and B, and to assign the sum to the floating point number represented by the variable name C.

The plus sign (+) specifies addition. Standard computer jargon refers to the plus sign an "operator." + is an arithmetic operator, because it specifies addition, which is an arithmetic operation.

Arithmetic operators are easy enough to understand; we all learn to add, subtract, multiply, and divide in early childhood. But there are two other types of operators: relational operators and Boolean operators. These are also easily understood, but they take a little more explanation, since they do not reflect day to day experiences.

Table 4-2 summarizes the BASIC operators. We will examine each group of operators in turn, beginning with arithmetic operators.

Table 4-2. Operators

	Precedence	Operator	Meaning
	High 9	( )	Parentheses denote order of evaluation
Arithmetic Operators	8	↑	Exponentiation
	7	−	Unary Minus
	6	•	Multiplication
	6	/	Division
	5	+	Addition
	5	−	Subtraction
Relational Operators	4	=	Equal
	4	< >	Not equal
	4	<	Less than
	4	>	Greater than
	4	< = or = <	Less than or Equal
	4	> = or = >	Greater than or Equal
Boolean Operators	3	NOT	Logical complement
	2	AND	Logical AND
	1 Low	OR	Logical OR

Arithmetic Operators

An arithmetic operator specifies addition, subtraction, multiplication, division, or exponentiation. Arithmetic operations are performed using floating point numbers. Integers are automatically converted to floating point numbers before an arithmetic operation is performed; the result is automatically converted back to an integer, if an integer variable represents the result.

The data operated on by any operator is referred to as an "operand." Arithmetic operators each require two operands, which may be numbers and/or numeric variables.

Addition (+). The plus sign specifies that the data (or operand) on the left of the + sign must be added to the data (or operand) on the right. For numeric quantities this is straightforward addition. Examples:

```
2+2
A+B+C
X%+1
BR+10E-2
```

The plus sign (+) is also used to "add" strings; but rather than adding their values, they are joined together, or concatenated, to form one longer string. The difference between numeric addition and string concatenation can be visualized as follows:

```
Addition of Numbers:
  num1+num2=num3

Addition of Strings:
  string1+string2=string1string2
```

Via concatenation, strings containing up to 255 characters can be developed. Examples:

"FOR"+"WARD"	results in "FORWARD"
"HI"+" "+"THERE"	results in "HI THERE"
A$+B$	results in concatenation of the two strings represented by string variable labels A$ and B$
"1" + CH$+E$	results in the character "1," followed by concatenation of the two strings represented by string variable labels CH$ and E$

In the illustrations above, if A$ is set equal to "FOR" and B$ is set equal to "WARD," then A$ + B$ would generate the same results as "FOR" + "WARD."

Subtraction (−). The minus sign specifies that the data (or operand) to the right of the minus sign is to be subtracted from the data (or operand) to the left of the minus sign. Examples:

4-1	results in 3
100-64	results in 36
A-B	results in the variable represented by label B being subtracted from the variable represented by label A
55-142	results in -87

In the example above, if A is assigned the value 100, and B is assigned the value 64, then the second and third examples are identical.

The minus operator is also used to identify a negative number. Examples:

-5	
-9E4	
-B	
4--2	Note that 4--2 is the same as 4+2

Multiplication (*). An asterisk specifies that the data (or operand) on the right of the asterisk is multiplied by the data (or operand) on the left of the asterisk. Examples:

100*2	results in 200
50*0	results in 0
A*X1	results in multiplication of two floating point numbers represented by floating point variables labeled A and X1
R%*14	results in an integer represented by integer variable label R% being multiplied by 14

In the examples above, if variable A is assigned the value 4.2, and variable X1 is assigned the value 9.63, then the illustrated multiplication would generate 40.446. A and X1 could hold integer values 100 and 2 to duplicate the first example; however the two numbers would be held in the floating point format as 100.0 and 2.0, since A and X1 are floating point variables. In order to multiply 100 by 2, representing these numbers as integers, the example would have to be A%*X1%.

Division (/). The slash specifies that the data (or operand) on the left of the slash is to be divided by the data (or operand) on the right of the slash. Examples:

10/2	results in 5
6400/4	results in 1600
A/B	results in the floating point number assigned to variable A being divided by the floating point number assigned to variable B
4E2/XR	results in 400 being divided by the floating point number represented by label XR

The third example, A/B, can duplicate the first or second example, even though A and B represent floating point numbers. But the integer numbers would be held in floating point form. A%/B% could exactly duplicate either of the first two examples, however.

Exponentiation (↑). The up arrow specifies that the data (or operand) on the left of the up arrow is raised to the power specified by the data (or operand) on the right of the up arrow. If the data (or operand) on the right is 2, the number on the left is squared; if the data (or operand) on the right is 3, the number on the left is cubed, etc. The exponent can be any number, variable, or expression, as long as the exponentiation yields a number in the allowed floating point range. Examples:

2↑2	results in 4
12↑2	results in 144
1↑3	results in 1
A↑5	results in the floating point number assigned to variable A being raised to the 5th power
2↑6.4	results in 84.4485064
NM↑-10	results in the floating point number assigned to variable NM being raised to the negative 10th power
14↑F	results in 14 being raised to the power specified by floating point variable F

Order of Evaluation

An expression may have multiple arithmetic operations, as in the following statement:

```
A+C*10/2↑2
```

When this occurs, there is a fixed sequence in which operations are processed. **First comes exponentiation (↑), followed by sign evaluation, followed by multiplication and division (*/), followed by addition and subtraction (+ −).** Operations of the same hierarchy are evaluated from left to right. **This order of operation can be overridden by the use of parentheses. Any operation within parentheses is performed first.** Examples:

4+1*2	results in 6
(4+1)*2	results in 10
100*4/2-1	results in 199
100*(4/2-1)	results in 100
100*(4/(2-1))	results in 400

When parentheses are present, CBM BASIC evaluates the innermost set first, then the next innermost, etc. Parentheses can be nested to any level, and may be used freely to clarify the order of operations being performed in an expression.

Relational Operators

Relational operators represent the conditions: greater than (>), less than (<), equal (=), not equal (<>), greater than or equal (> =), and less than or equal (< =).

1=5-4	results in true (-1)
14>66	results in false (0)
15>=15	results in true (-1)
A<>B	the result will depend on the values assigned to floating point variables A and B

CBM BASIC arbitrarily assigns a value of 0 to a "false" condition; a value of −1 is assigned to a "true" condition. These 0 and −1 values can be used in equations. For example, in the expression (1=1)*4, (1=1) is true. True equates to −1, therefore the expression is the same as (−1)*4, which results in −4. You can include any relational operators within a CBM BASIC expression. Here are some more examples:

```
25+(14>66)              is the same as    25+0
(A+(1=5-4))•(15>=15)    is the same as    (A-1)•(-1)
```

Relational operators can be used to compare strings. For comparison purposes, the letters of the alphabet have the order A<B, B<C, C<D, etc. Strings are compared one character at a time, starting with the leftmost character. Examples:

```
"A"<"B"       results in true (-1)
"X"="XX"      results in false (0)
C$=A$+B$      the result will depend
                on the string values assigned
                to the three string variables
                C$, B$, and A$
```

When operating on strings, as for numbers, CBM BASIC generates a value of −1 if a relational operator specifies a "true" condition; a value of 0 is generated for a "false" condition. Here are some examples:

```
("JONES">"DOE")+37                 is the same as    -1+37
("AAA"<"AA")•(Z9-("OTTER">"AB"))   is the same as    0•(Z9-(-1))
```

Boolean Operators

Boolean operators give programs the ability to make logical decisions. There are four standard Boolean operators: AND, OR, EXCLUSIVE OR, and NOT. **CBM BASIC supports** three of these operators: **AND, OR, and NOT.**

If you do not understand Boolean operators, then a simple supermarket shopping analogy will serve to illustrate Boolean logic.

Suppose you are shopping for breakfast cereals with two children.

The AND Boolean operator says that a cereal is selected if child A *and* child B select the cereal.

The OR Boolean operator says that a cereal will be selected if either child A *or* child B selects the cereal.

The NOT operator generates an opposite. If child B insists on disagreeing with child A, then child B's decision is always the *not* of child A's decision.

Computers do not work with analogies; they work with numbers. Therefore Boolean logic reduces all variables and results to 0 or 1. **Table 4-3 summarizes the way in which Boolean operators handle numbers. This table is referred to as a "truth table."**

Boolean operators are used to control program execution logic; here are some examples:

```
IF A=100 AND B=100 GOTO 10
  If both A and B are equal to 100, branch to line 10
IF X < Y AND B >=44 THEN F=0
  If X is less than Y, and B is greater than or equal to 44,
  then set F equal to 0
IF A=100 OR B=100 GOTO 20
  If either A or B has a value of 100, branch to line 20.
IF X<Y OR B>=44 THEN F=0
  F is set to 0 if X is less than Y, or B is greater than 43
IF A=1 AND B=2 OR C=3 GOTO 30
  Take the branch if both A=1 and B=2; also take
  the branch if C=3
```

A single operand can be tested for "true" or "false." An operand appearing alone has an implied "< >0" following it. Any non-zero value is considered true; a zero value is considered false.

```
IF A THEN B=2
IF A< >0 THEN B=2
  The above two statements are equivalent
IF NOT B GOTO 100
  Branch if B is false, i.e., equal to zero. This is
    probably better written as:
    IF B=0 GOTO 100
```

All Boolean operations use integer operands. If you perform Boolean operations using floating point numbers, then the numbers are automatically converted to integers; therefore the floating point numbers must fall within the allowed range of integer numbers.

You cannot perform Boolean operations using string operands.

If you are a beginning programmer, you are unlikely to use Boolean operators in the manner which we are about to describe. If you find you do not understand the discussion, then skip to the next section.

Table 4-3. Boolean Truth Table

The AND operation results in a 1 only if both bits are 1

1 AND 1 = 1
0 AND 1 = 0
1 AND 0 = 0
0 AND 0 = 0

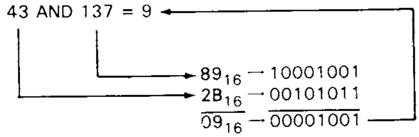
The OR operation results in a 1 if either bit is 1

1 OR 1 = 1
0 OR 1 = 1
1 OR 0 = 1
0 OR 0 = 0

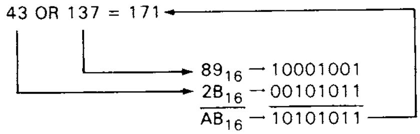
The NOT operation logically complements each bit

NOT 1 = 0
NOT 0 = 1

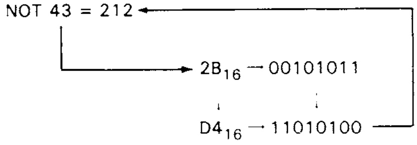
Boolean operators operate on integer operands one binary digit at a time. CBM BASIC stores all numbers in binary format, using two's complement notation to represent negative numbers. Therefore we can illustrate an AND operation as follows:

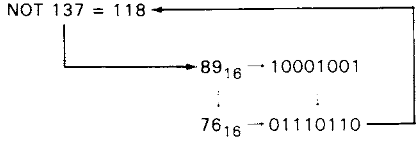


Here is an OR operation:



Here are two NOT operations:





Boolean operations of this type are used in engineering applications.*

If operands are not integers, they are converted to integer form; the Boolean operation is performed, and the result is returned as a 0 or 1.

If a Boolean operator has relational operands, then the relational operand is evaluated to −1 or 0 before the Boolean operation is performed. Thus the operation:

```
A=1 OR C<2
```

is equivalent to:

$$\left\{\begin{matrix}-1\\ \text{or}\\ 0\end{matrix}\right\}\ \text{OR}\ \left\{\begin{matrix}-1\\ \text{or}\\ 0\end{matrix}\right\}$$

Consider this more complex operation:

```
IF A=B AND C<D GOTO 40
```

First the relational expressions are evaluated. Assume that the first expression is true and the second one is false. In effect, the following Boolean expression is evaluated as follows:

```
IF -1 AND 0 GOTO 40
```

---

*If you wish to learn more about binary arithmetic and Boolean operations, see *An Introduction to Microcomputers: Volume 0 — The Beginners Book* by A. Osborne, Osborne/McGraw-Hill, 1977.

Performing the AND yields a 0 result:

```
IF 0 GOTO 40
```

Recall that a single term has an implied "< >0" following it. The expression therefore becomes:

```
IF 0 < > GOTO 40
```

Thus, the branch is not taken.

In contrast, a Boolean operation performed on two variables may yield any integer number:

```
IF A% AND B% GOTO 40
```

Assume that A%=255 and B%=240. The Boolean operation 255 AND 240 yields 240. The statement, therefore, is equivalent to:

```
IF 240 GOTO 40
```

or, with the "< >0":

```
IF 240 < > 0 GOTO 40
```

Therefore the branch will be taken.

Now compare the two assignment statements:

```
A = A AND 10
A = A <10
```

In the first example, the current value of A is logically ANDed with 10 and the result becomes the new value of A. A must be in the integer range −32767 to +32768. In the second example, the relational expression A<10 is evaluated to −1 or 0, so A must end up with a value of −1 or 0.

ARRAYS

Arrays are used frequently, in every type of computer program. If you do not understand arrays, then you must learn about them. The information that follows will be very important to your programming efforts.

Conceptually, arrays are very simple. When you have two or more related data items, instead of giving each data item a separate variable name, you give the collection of related data items a single variable name. Then you select individual items using a position number, which in computer jargon is referred to as a subscript, an index, or a dimension.

A grocery list, for example, may have six items from the meat and poultry department, four fruit and vegetable items, three dairy products, etc. These three groups of items could each be represented by a single variable name as follows:

```
MP$(0) = "CHOPPED SIRLOIN"    FV$(0) = "ORANGES"
MP$(1) = "CHUCK STEAK"        FV$(1) = "APPLES"
MP$(2) = "NEW YORK STEAK"     FV$(2) = "BEANS"
MP$(3) = "CHICKEN"            FV$(3) = "CARROTS"
MP$(4) = "SALAMI"
MP$(5) = "SAUSAGES"           DP$(0) = "MILK"
                              DP$(1) = "CREAM"
                              DP$(2) = "COTTAGE CHEESE"
```

MP$ is a single variable name that identifies all meat and poultry products.
FV$ identifies fruits and vegetables, while DP$ identifies dairy products.

A subscript (index or dimension) follows each variable name. Thus a specific data item is identified by a variable name and an index.

We could take the array concept one step further, specifying a single variable name for the entire grocery list, using two indexes. The first index (or dimension) specifies the product type and the second index (or dimension) specifies the item within the product type. This is one way in which a single grocery list variable array with two subscripts could replace the three arrays with single subscripts illustrated above:

```
GL$(0,0) = MP$(0)    GL$(1,0) = FV$(0)    GL$(2,0) = DP$(0)
GL$(0,1) = MP$(1)    GL$(1,1) = FV$(1)    GL$(2,1) = DP$(1)
GL$(0,2) = MP$(2)    GL$(1,2) = FV$(2)    GL$(2,2) = DP$(2)
GL$(0,3) = MP$(3)    GL$(1,3) = FV$(3)
GL$(0,4) = MP$(4)
GL$(0,5) = MP$(5)
```

Arrays can represent integer variables, floating point variables, or string variables; however, a single array variable can only represent one data type. In other words, a single variable cannot mix integer and floating point numbers. One or the other can be present, but not both.

Arrays are a useful shorthand means of describing a large number of related variables. Consider, for example, a table of numbers containing ten rows of numbers, with twenty numbers in each row. There are 200 numbers in the table. How would you like it if you had to assign a unique name to each of the 200 numbers? It would be far simpler to give the entire table one name, and identify individual numbers within the table by their table location. That is precisely what an array does for you.

Arrays can have one or more dimensions. An array with a single dimension is equivalent to a table with just one row of numbers. The dimension identifies a number within the single row. (Engineers use the word "vector" to describe an array with a single dimension.) An array with two dimensions yields an ordinary table with rows and columns: one dimension identifies the row, the other dimension identifies the column. An array with three dimensions yields a "cube" of numbers, or perhaps a stack of tables. Four or more dimensions yield an array that is hard to visualize, but mathematically no more complex than a smaller-dimensioned array.

Let us examine arrays in detail.

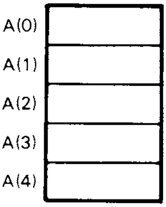
A single-dimensional array element has the form:

name(i)

where:

name	is the variable name for the array. Any type of variable name may be used.
	is the array index to that element. i must start at 0.

A single-dimensional array called A, having five elements, can be visualized as follows:



The number of elements in the array is equal to the highest index number, plus 1. This takes array elements 0 into account.

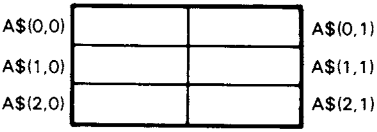
A two-dimensional array element has the form:

name(i,j)

where:

name	is the variable name of the array
i	is the column index
j	is the row index

A two-dimensional array called A$, having three column elements and two row elements, might be visualized as follows:



The size of the array is the product of the highest row dimension plus 1, multiplied by the highest column dimension plus 1. For the array above, it is 3×2=6 elements.

Additional dimensions can be added to the array:

name (i,j,k,. . .)

Arrays of up to eleven elements (index 0 to 10 for a single dimensioned array) may be used routinely in CBM BASIC. Arrays containing more than eleven elements need to be "declared" in a Dimension statement. Dimension statements are described later in this chapter. An array (always with subscripts) and a single variable of the same name are treated as separate items by CBM BASIC.

BASIC COMMANDS

In Chapters 2 and 3 we describe a number of commands which you enter via the keyboard in order to control CBM computer operations. RUN is one such command. **Commands can all be executed as BASIC statements.**

You are unlikely to execute commands out of BASIC statements when you first start writing programs.

When you start writing very large programs you will run out of memory space. Then you must break a program up into a number of smaller modules and execute them one at a time. Each module must load the next module in turn. This is described in Chapter 6.

Reserved Words

All of the character combinations that define a BASIC statement's operations, and all functions, are called "reserved words." **Table 4-4 lists all CBM BASIC reserved words.** You will have encountered many of these reserved words in this chapter, but others are not described until Chapter 6.

Table 4-4. Reserved Words

WORD	Abbreviations		WORD	Abbreviations		WORD	Abbreviations		WORD	Abbreviations	
	Alternate Character Set	Standard Character Set		Alternate Character Set	Standard Character Set		Alternate Character Set	Standard Character Set		Alternate Character Set	Standard Character Set
ABS	aB	A│	DS$*	ds$	DS$	NEW	new	NEW	SCRATCH*	sC	S─
AND	aN	A╱	DSAVE*	dS	D♥	NEXT	nE	N‾	SGN	sG	S│
APPEND*	aP	A┐	END	eN	E╱	NOT	nO	N┌	SIN	sI	S╲
ASC	aS	A♥	EXP	eX	E♣	ON	on	ON	SPC(	sP	S┐
ATN	aT	A│	FN	fn	FN	OPEN	oP	O┐	SQR	sQ	S●
BACKUP*	bA	B♠	FOR	fO	F┌	OR	or	OR	ST	st	ST
CHR$	cH	C│	FROM	fR	F_	PEEK	pE	P‾	STATUS	status	STATUS
CLOSE	clO	CL┌	GET	gE	G‾	POKE	pO	P┌	STEP	stE	ST‾
CLR	cL	CL	GET #	get#	GET#	POS	pos	POS	STOP	sT	S│
CMD	cM	C╲	GOTO	gO	G┌	PRINT	?	?	STR$	str$	STR$
COLLECT*	coL	COL	GOSUB	goS	GO♥	PRINT #	pR	P_	SYS	sY	S│
CONCAT*	conC	CON─	HEADER*	hE	H‾	READ	rE	R‾	TAB(	tA	T♠
CONT	cO	C┌	IF	if	IF	READ #	read#	READ#	TAN	tan	TAN
COPY*	coP	CO┐	INPUT	input	INPUT	RECORD*	reC	RE─	THEN	tH	T│
COS	cos	COS	INPUT#	iN	I╱	REM	rem	REM	TI	ti	TI
DATA	dA	D♠	INT	int	INT	RENAME*	reN	RE╱	TIME	time	TIME
DCLOSE*	dC	D─	LEFT$	leF	LE─	RESTORE	reS	RE♥	TI$	ti$	TI$
DEF	dE	D‾	LEN	len	LEN	RETURN	reT	RE│	TO	to	TO
DIM	dI	D╲	LET	lE	L‾	RIGHT$	rI	R╲	US	uS	U♥
DIRECTORY*	diR	DI_	LIST	lI	L╲	RND	rN	R╱	VAL	vA	V♠
DLOAD*	dL	DL	LOAD	lO	L┌	RUN	rU	R╱	VERIFY	vE	V‾
DOPEN*	dO	D┌	LOG	log	LOG	SAVE	sA	S♠	WAIT	wA	W♠
DS*	ds	DS	MID$	mI	M╲						

* These are reserved words in BASIC versions 4.0 and higher only.

When executing BASIC programs, the CBM computer scans every BASIC statement, seeking out any character strings that constitutes a reserved word. The only exception is text strings enclosed in quotes. This can cause trouble if a reserved word is embedded anywhere within a variable name. The CBM computer is not smart enough to identify a variable name by its location in BASIC statement. **Therefore you should be very careful to keep reserved words out of your variable names;** this is particularly important with the short reserved words that can easily slip into a variable name.

Some reserved words are shown in Table 4-4 with an asterisk. These reserved words apply only to CBM BASIC versions 4.0 and higher. Nevertheless it is a good idea not to use these reserved words in any CBM BASIC program. You never know when you may wish to upgrade a program so that it runs on a newer CBM computer using BASIC 4.0.

BASIC Word Abbreviations

You learned early in this book that the BASIC statement PRINT could always be entered from the keyboard by the abbreviation ?, the question mark character. ? is expanded by the CBM BASIC interpreter to the full word PRINT.

Most BASIC commands, statements, and functions can be abbreviated using the first two characters of the keyword, with the second character entered in shifted mode. With the standard character set, the second character appears as a graphic character. For example, the abbreviation for LIST appears as:

```
      L╲
or    lI
```

Where a two-letter abbreviation is ambiguous (does ST mean STEP or STOP?) the two-letter abbreviation is assigned to the most frequently used keyword, and the other word (or words) are either not abbreviated or are abbreviated by the first three characters with the third entered in shifted mode. For STEP/STOP, STOP is abbreviated:

```
                              sT
                         or   S|
```

STEP is abbreviated:

```
                              stE
                         or   ST‾
```

To abbreviate STEP, type unshifted S (capital S), unshifted T (capital T), and shifted E (graphic 3/4 TOP LINE HORIZONTAL).

Following are a few sample input lines showing use of the two- and three-letter abbreviations wherever possible. All the abbreviated words are expanded to the full spelling when you list the programs.

```
pO 59468,14                    (after RETURN) Abbreviation for POKE
10 lE a=10
20 b=a aN 14+eX(2)
30 dI c(5)
40 fO i=0 to 5
50 rE c(i)
60 nE
70 dA 1,6,2,4,10,5,16
80 reS
90 eN
lI                             Abbreviation for LIST
10 let a=10
20 b=a and 14+exp(2)
30 dim c(5)
40 for i=0 to 5
50 read c(i)
60 next
70 data 1,6,2,4,10,5,16
80 restore
90 end
pO 59468,12                    (before RETURN) Abbreviation for POKE
```

After keying RETURN at the last POKE statement line (return to Standard Character Set), you will see the abbreviations show with graphics as the shifted characters, and the expanded listing will display upper case letters.

A list of reserved words and their abbreviations, if any, is given in Table 4-4. Note that the expansions from abbreviations for the two functions SPC and TAB include the left parenthesis. This means that if you use the abbreviation for either of these, you must not type in the left parentheses. For example:

```
10 ?sP(5)
```

expands to:

```
10 print spc((5)
```

syntax error results from two left parentheses

The correct keyin is:

```
10 ?sP5)
```

This parenthesis rule applies only to the SPC and TAB functions and is a format inconsistency you will have to watch for when abbreviating these function names. For all other functions, you key in both parentheses. For example:

```
10 ?rN(1)
```

BASIC STATEMENTS

The operation performed by a statement is specified using "reserved words" (see Table 4-4).

Remember, Chapter 8 provides a complete description of every statement recognized by CBM BASIC. This chapter introduces you to programming concepts, stressing the way statements are used. No statement is described in detail in this chapter. Read the statement description given in Chapter 8 if you do not understand how any statement is being used.

REMARKS

It is appropriate that any discussion of BASIC statements begins by describing the only BASIC statement which the computer will ignore: the remark. If the first three characters of a BASIC statement are REM, then the computer ignores the statement entirely. So why include such a statement? The answer is that remarks make your program easier to read.

If you write a short program with five or ten statements, you will probably have little trouble remembering what the program does — unless you leave it around for six months and then try to use it again. If you write a longer program with 100 or 200 statements, then you are quite likely to forget something very important the very next time you use the program. After you have written dozens of programs, you will stand no chance of remembering each program in detail. The solution to this problem is to document your program by including remarks that describe what is going on.

Good programmers use plenty of remarks in all of their programs. In all of this chapter's program examples we will include remarks that describe what is going on, simply to get you into the habit of doing the same thing yourself.

Remark statements have line numbers, like any other statement. A remark statement's line number can be used like any other statement line number.

ASSIGNMENT STATEMENT

Assignment statements let you assign values to variables. You will encounter assignment statements frequently, in every type of BASIC program. Here are some examples of assignment statements:

```
90 REM INITIALIZE VARIABLE X
100 LET X=3.24
```

In statement 100, floating point variable X is assigned the value 3.24

```
150 X=3.24
```

Equivalent to statement 100 above; the LET is optional in all assignment statements

```
215 A$="ALSO RAN"
```

The string variable A$ is assigned the two text words ALSO RAN

Here are three assignment statements that assign values to array variable DP$(I), which we encountered earlier when describing arrays:

```
200 REM DP$(I) IS THE DAIRY PRODUCTS SHOPPING
    LIST VARIABLE
210 DP$(0)="MILK"
220 DP$(1)="CREAM"
230 DP$(2)="COTTAGE CHEESE"
```

Remember, we can put more than one statement on a single line; therefore the three DP$ assignments could be placed on a single line as follows:

```
200 REM DP$(I) IS THE DAIRY PRODUCTS SHOPPING
    LIST VARIABLE
210 DP$(0)="MILK":DP$(1)="CREAM":DP$(2)=
    "COTTAGE CHEESE"
```

Recall that a colon must separate adjacent statements appearing on the same line.

Assignment statements can include any of the arithmetic or relational operators described earlier in this chapter. Here is an example of such an assignment statement:

```
90 REM THIS IS A DUMB WAY TO ASSIGN A VALUE TO V
100 V=3.24+7.96/8.5
```

This statement assigns the value 4.17647059 to floating point variable V; it is equivalent to these three statements:

```
90 REM X AND Y NEED TO BE INITIALIZED SEPARATELY
   FOR LATER USE
100 X=7.96
110 Y=8.5
120 V=3.24+X/Y
```

which could be written on one line as follows:

```
100 X=7.96:Y=8.5:V=3.24+X/Y
```

Here are assignment statements that perform the Boolean operations given earlier in this chapter:

```
90 REM THESE EXAMPLES WERE DESCRIBED EARLIER IN THE
   CHAPTER
100 A%=43 AND 137
200 B%=43 OR 137
```

The following example shows how a string variable could have its value assigned using string concatenation:

```
100 V$="COTTAGE"
200 W$="CHEESE"
300 DP$(2)=V$+" "+W$
400 REM DP$(2) IS ASSIGNED THE STRING VALUE "COTTAGE CHEESE"
```

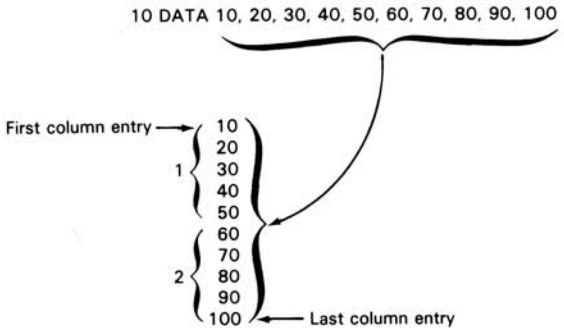
DATA and READ Statements

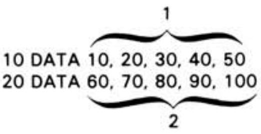
When a number of variables need data assignments, the DATA and READ statements should be used rather than the LET statement. Consider the following example:

```
5 REM INITIALIZE ALL PROGRAM VARIABLES
10 DATA 10,20,-4,16E6
20 READ A,B,C,D
```

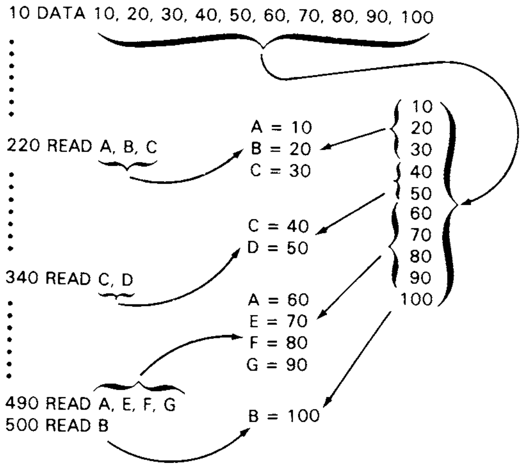
The statement on line 10 specifies four numeric data values. These four values are assigned to four floating point variables by the statement on line 20. After statements on lines 10 and 20 have been executed, $A=10$, $B=20$, $C=-4$ and $D=16\times10^6$.

If you have one or more DATA statements in your program, then you can visualize them as building a "column" of numbers. For example, a DATA statement that contains a list of 10 numbers would build a ten-entry column. Two DATA statements each specifying five of the ten data entries would build exactly the same column. This may be illustrated as follows:



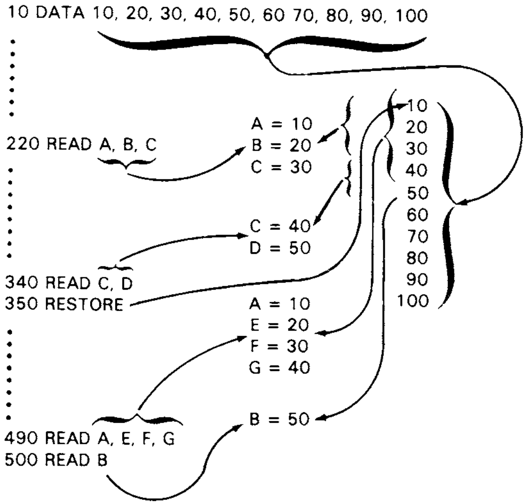


The first READ statement in the program starts at the first column entry and takes numbers sequentially, assigning them to variables named in the READ statement. The second (and subsequent) READ statements take values from the column, starting at the point where the previous READ statement left off. This may be illustrated as follows:



RESTORE Statement

You can at any time send the pointer back to the beginning of the numeric column by executing a RESTORE statement. Here is an example of the use of RESTORE:



DIMENSION STATEMENT

CBM BASIC normally assumes an array variable has a single dimension, with index values of 0 through 10. This generates an eleven-element array. **If you want a single dimension with more, or less, than eleven elements, then you must include the array variable in a dimension statement. You must include the array in a dimension statement if it has two or more dimensions, whatever number of elements the array may have.** The following example provides dimensions for the three single-indexed variables MP$, FV$, and DP$. We used these variables in our earlier discussion of arrays.

```
DIM MP$(5),FV$(3),DP$(2)
```

The double-dimension grocery list variable would be dimensioned as follows:

```
DIM GL$(3,5)
```

A dimension statement can provide dimensions for any number of variables, providing the statement fits within an 80-column line.

The number (or numbers) following a variable name in a DIM statement is equal to the largest index value that can occur in that particular index position. But remember indexes begin at 0. Therefore MP$(5) dimensions the variable MP$ to have six values, not five, since indexes 0, 1, 2, 3, 4, and 5 will be allowed. GL$ (3,5), likewise, specifies a double-dimension variable with 24 entries, since the first dimension can have values 0, 1, 2, and 3, while the second dimension can have values 0 through 5.

Once you have specified an array variable in a dimension statement, you must subsequently reference the variable with the specified number of indexes; each index must have a value between 0 and the number specified in the dimension statement. If any of these syntax rules are broken a syntax error will be reported.

BRANCH STATEMENTS

Statements within a BASIC program are normally executed in ascending order of line numbers. This execution sequence was explained earlier in this chapter when we described line numbers. Branch statements change this execution sequence.

GOTO Statement

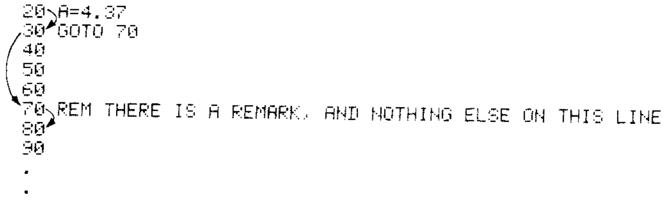
GOTO is the simplest branch statement; it **allows you to specify the statement which will be executed next.** Consider the following example:

```
20 A=4.37
30 GOTO 100
40
50
60
70
80
90
100
110
 .
 .
```

The statement on line 20 is an assignment statement; it assigns a value to floating point variable A. The next statement is a GOTO; it specifies that program execution must branch to line 100. Therefore the instruction execution sequence surrounding this part of the program will be line 20, then line 30, then line 100.

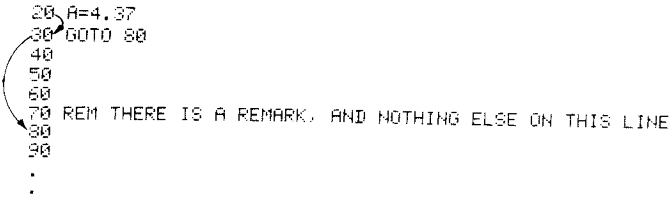
Of course, some other statement must branch back to line 40, otherwise the statement on line 40 would never be executed by program logic as illustrated above.

You can branch to any line number, even if the line has nothing but a remark on it. However, the computer ignores the remark, so the effect is the same as branching to the next line. For example, consider the following branch:



```
20 A=4.37
30 GOTO 70
40
50
60
70 REM THERE IS A REMARK, AND NOTHING ELSE ON THIS LINE
80
90
.
.
```

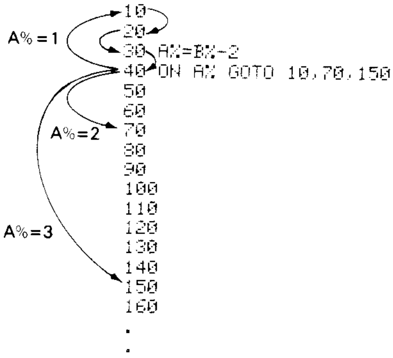
Program execution branches from line 30 to line 70; there is nothing but a remark on line 70, therefore the computer moves on to line 80, executing statements on this line. Therefore, even though you can branch to a remark, you might as well branch to the next line. This may be illustrated as follows:



```
20 A=4.37
30 GOTO 80
40
50
60
70 REM THERE IS A REMARK, AND NOTHING ELSE ON THIS LINE
80
90
.
.
```

Computed GOTO Statement

There is also a computed GOTO statement that lets program logic branch to one of two or more different line numbers, depending on the current value of a variable. Consider the following illustration:



```
10
20
30 A%=B%-2
40 ON A% GOTO 10,70,150
50
60
70
80
90
100
110
120
130
140
150
160
.
.
```

The statement on line 40 is a computed GOTO. When this statement is executed, program logic will branch to statement 10 if variable A% = 1, the branch will be to statement 70 if variable A%=2, while A%=3 causes a branch to statement 150. If A% has any other value than 1, 2, or 3, an error is reported. Notice that variable A% is assigned a value in statement 30. The value assigned to A% depends on the current value of variable B%. The illustration does not show how variable B% is computed; however, so long as B% has a value of 3, 4, or 5, the statement on line 40 will cause a branch to be taken.

To test the computed GOTO statment, key in the following program:

```
10 B%=4
20 ?B%
30 A%=B%-2
40 ON A%GOTO 10,70,150
70 ?B%
80 B%=5
90 GOTO 30
150 ?B%
160 B%=3
170 GOTO 20
```

Now execute this program by typing RUN on any blank line. Do not type RUN on any line that already is displaying something. If you do, you will get a syntax error and the program will not be executed.

Can you account for the sequence in which digits are displayed? Try rewriting the program so that each number is displayed once, in the sequence: 345345345...

LOOPED CONTROL STATEMENTS

FOR-NEXT Statement

GOTO and computed GOTO statements let you create any type of statement execution sequence that your program logic may require. But **suppose you want to re-execute an instruction, (or a group of instructions) many times.** For example, suppose array variable A(I) has 100 elements and each element needs to be assigned a value ranging from 0 to 99. Writing a hundred assignment statements would be very tedious. It is far simplier to re-execute one statement one hundred times. **This can be done using the FOR and NEXT statements** as follows:

```
10 DIM A(99)
20 FOR I=0 TO 99 STEP 1
30 A(I)=I
40 NEXT I
```

Statement(s) between FOR and NEXT are executed repeatedly. In this case a single assignment statement appears between FOR and NEXT; therefore this single statement is re-executed repeatedly.

In order to test the workings of FOR-NEXT loops, we will display A(I) values created within the loop. Key in the following program:

```
10 DIM A(99)
20 FOR I=0 TO 99 STEP 1
30 A(I)=I
35 ?A(I);
40 NEXT I
50 REM IF YOU HAVE A GOTO STATEMENT THAT BRANCHES TO ITSELF, THE
70 REM COMPUTER EXECUTES AN ENDLESS LOOP, IN EFFECT, IT WAITS
80 GOTO 80
```

Now key in RUN. The program is executed. One hundred numbers are displayed, starting at 0 and ending at 99. Press the STOP key to stop program execution.

Statements between FOR and NEXT are re-executed the number of times specified by the index variable appearing directly after FOR; in the illustration above this index variable is I. I is specified as going from 0 to 99 in increments of 1. I also appears in the assignment statement. Therefore the first time the assignment statement is executed, I will equal 0 and the assignment statement will be executed as follows:

```
30 A(0)=0
```

I is increased by the step, or increment, size, which is specified on line 20 as 1; I therefore equals the second time the assignment statement on line 30 is executed. The assignment statement has effectively become:

```
30 A(1)=1
```

I continues to be incremented by the specified STEP until the maximum value of 99 is reached or exceeded.

STEP does not have to be 1; it can have any integer value. Change step to 5 on line 20 and re-execute the program. Now the assignment statement is executed just 20 times, since incrementing I by 5 nineteen times will take it to 95; the 20th increment will take it to 100, which is more than the maximum value of 99. Keeping STEP at 5, we could allow the assignment statement to be executed 100 times by increasing the maximum value of I to 500. Can you make this change? (Remember to change the dimension statement as well.)

The step size does not have to be positive. But if the step size is negative, then the initial value of I must be larger than the final value of I. For example if the step size is −1, and we want to initialize 100 elements of AC(I) with values ranging from 0 to 99, then we would have to rewrite the statement on line 20 as follows:

```
10 DIM A(99)
20 FOR I=99 TO 0 STEP -1
30 A(I)=I
35 ?A(I);
40 NEXT I
80 GOTO 80
```

Execute this program to test the negative STEP.

The initial and final values for I, and the step size, are evaluated as integers; but no other restrictions are placed on these three values. You can specify these three values using floating point variables or expressions. Expressions will be evaluated to a floating point result. Then the floating point result will be converted to an integer using the round-off rules described earlier in this chapter.

Because round-off rules can cause problems, you are strongly urged to specify beginning, ending and step sizes as integers. Do not use expressions since this unnecessarily complicates the program. If you must calculate one of these values, it is simplier and faster to do so in a separate statement.

If the step size is 1 (and this is frequently the case), **you do not have to include a step size definition.** In the absence of any definition, CBM BASIC assumes a step size of 1. Therefore we could rewrite the statement on line 20 as follows:

```
10 DIM A(99)
15 REM USE A STEP SIZE OF 1
20 FOR I=0 TO 99
30 A(I)=I
35 ?A(I);
40 NEXT I
80 GOTO 80
```

Also, you do not need to specify the index variable in the NEXT statement. But if you do, it will make your program easier to read.

Nested Loops

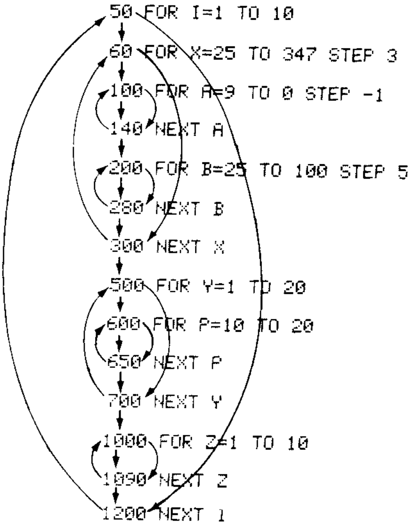
The FOR-NEXT structure is referred to as a "program loop" since statement execution loops around from FOR to NEXT, and back to FOR. This loop structure is very common; almost every BASIC program that you write will include one or more such loops. Loops are so common that they are frequently nested. **The statement sequence occurring between FOR and NEXT can be of any length; frequently it can run to tens or hundreds of statements. And within these tens or hundreds of statements, additional loops may occur.** The following illustration shows a single level of nesting:

```
10 DIM A(99)
20 FOR I=0 TO 99
30 A(I)=I
40 REM DISPLAY ALL VALUES OF A(I) ASSIGNED THUS FAR
50 FOR J=0 TO I
60 ?A(J)
70 NEXT J
80 NEXT I
90 GOTO 90
```

Complex loop structures appear frequently, even in relatively short programs. Here is an example, showing the FOR and NEXT statements, but none of the intermediate statements:

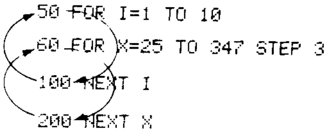
```
50 FOR I=1 TO 10
60 FOR X=25 TO 347 STEP 3
.
100 FOR A=9 TO 0 STEP -1
.
140 NEXT A
200 FOR B=25 TO 100 STEP 5
.
280 NEXT B
300 NEXT X
.
500 FOR Y=1 TO 20 STEP 2
.
600 FOR P=10 TO 20
.
650 NEXT P
700 NEXT Y
.
1000 FOR Z=1 TO 10
.
1090 NEXT Z
1200 NEXT I
```

The outermost loop uses index I; it contains three nested loops that use indexes X, Y, and Z. The first loop contains two additional loops which use indexes A and B. The second loop contains one nested loop using index P. The third loop contains no nested loops. **Each nested loop must have a different index variable name.** Statement execution sequences may be illustrated as follows:



```
50 FOR I=1 TO 10
60 FOR X=25 TO 347 STEP 3
100 FOR A=9 TO 0 STEP -1
140 NEXT A
200 FOR B=25 TO 100 STEP 5
280 NEXT B
300 NEXT X
500 FOR Y=1 TO 20
600 FOR P=10 TO 20
650 NEXT P
700 NEXT Y
1000 FOR Z=1 TO 10
1090 NEXT Z
1200 NEXT I
```

Loop structures are very easy to visualize and use. There is only one common error which you must avoid: **Do not terminate an outer loop before you terminate an inner loop.** For example, the following loop structure is illegal:



```
50 FOR I=1 TO 10
60 FOR X=25 TO 347 STEP 3
100 NEXT I
200 NEXT X
```

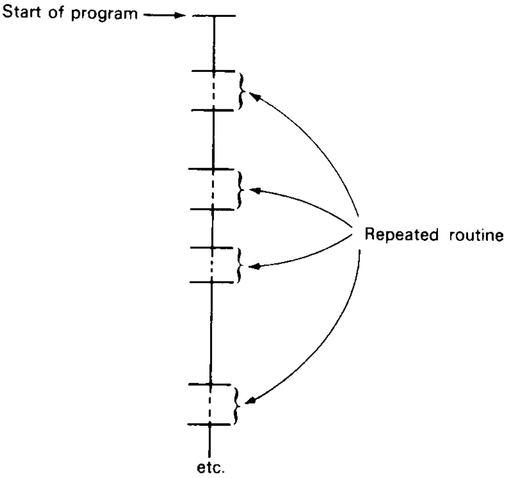
If you do not include the index variable in the NEXT statement, then program logic will automatically terminate loops correctly, since there is only one possible correct loop termination each time a NEXT statement is encountered. If you do not believe this, look again at the complex example illustrated earlier. Then work out some additional complex examples.

Every program must have the same number of FOR and NEXT statements, since every loop must begin with a FOR statement and end with a NEXT statement. For example, suppose there are two FOR statements, but only one NEXT statement. The second FOR statement constitutes an inner loop which will execute correctly. But the outer loop has no NEXT statement to terminate it and the program will execute incorrectly. If you have too many NEXT statements a syntax error will also be generated.

SUBROUTINE STATEMENTS

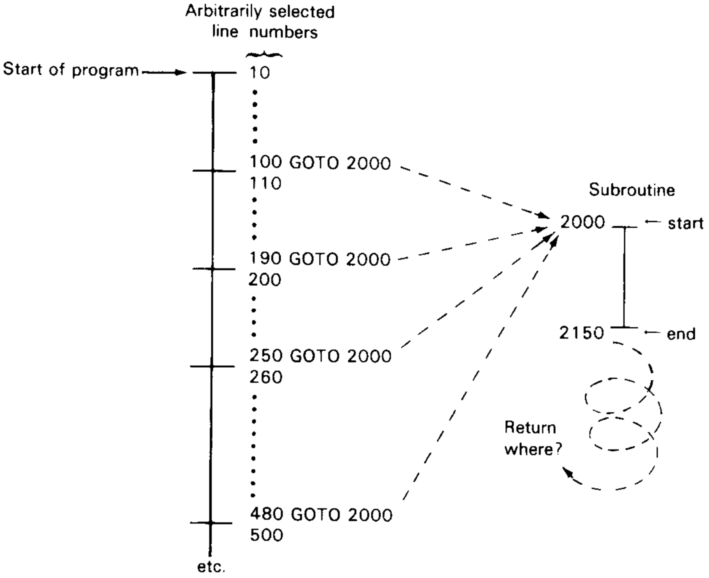
Once you start writing programs that are more than a few statements long, you will quickly find short routines that get used repeatedly. For example, suppose you have an array variable (such as A(I)) which is reinitialized frequently at different points in your program. Would you simply repeat the three instructions that constitute the FOR-NEXT loop that we described earlier? Since there are just three instructions, you may as well do so.

But suppose you have to initialize the array and then execute ten or eleven instructions that process array data in some fashion. If you had to use this loop many times within one program, rewriting ten to fifteen statements each time you wished to use the loop would take time; but more importantly it would waste a lot of computer memory. This may be illustrated as follows:



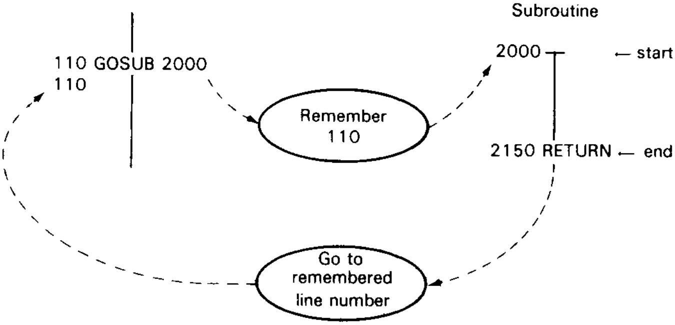
How about separating out the repeated statements and branching to them? That is precisely what we will do; **the group of statements is then referred to as a "subroutine."**

But a problem arises. Branching from your program to the subroutine is simple enough; the subroutine has an entry line number. But at the end of the subroutine, where do you branch back to? You *could* execute a GOTO statement whenever you wish to branch to a subroutine.



GOSUB Statement

At the end of the subroutine, where do you return to? If two GOTO statements branch to the subroutine, there are two different places to which you will wish to return after the subroutine has completed execution. The solution is to use special subroutine statements. **Instead of branching to the suboutine using a GOTO, use a GOSUB statement. This statement branches in the same way as a GOTO, but in addition it remembers the next line number.** This may be illustrated as follows:



End the subroutine with a RETURN statement. This statement causes a branch back to the line number which the GOSUB statement remembered. The three-statement loop which initializes array A(I) would appear as follows if it were converted into a subroutine:

```
10 REM MAIN PROGRAM
20 REM YOU CAN DIMENSION A SUBROUTINE'S VARIABLE IN THE MAIN
30 REM PROGRAM.  IT IS A GOOD IDEA TO DIMENSION ALL VARIABLES
50 REM AT THE START OF THE MAIN PROGRAM.
60 DIM A(99)
70 GOSUB 2000
80 REM DISPLAY SOMETHING TO PROVE THE RETURN OCCURRED
90 ?"RETURNED"
100 GOTO 100
2000 REM SUBROUTINE
2010 FOR I=0 TO 99
2020 A(I)=I
2030 ?A(I);
2040 NEXT I
2050 RETURN
```

Nested Subroutines

Subroutines can be nested. That is to say, a subroutine can itself call another subroutine, which in turn can call a third subroutine, and so on. You do not have to do anything special in order to use nested subroutines. Simply branch to the subroutine using a GOSUB statement and end the subroutine with a RETURN statement. CBM BASIC will remember the correct line number for each nested return. The following program illustrates nested subroutines:

```
10 REM MAIN PROGRAM
20 REM YOU CAN DIMENSION A SUBROUTINE'S VARIABLE IN THE MAIN
30 REM PROGRAM.  IT IS A GOOD IDEA TO DIMENSION ALL VARIABLES
50 REM AT THE START OF THE MAIN PROGRAM.
60 DIM A(99)
70 GOSUB 2000
80 REM DISPLAY SOMETHING TO PROVE THE RETURN OCCURRED
90 ?"RETURNED"
100 GOTO 100
2000 REM FIRST LEVEL SUBROUTINE
2010 FOR I=0 TO 99
2020 A(I)=I
2030 GOSUB 3000
2040 NEXT I
2050 RETURN
3000 REM NESTED SUBROTINE
3010 ?A(I)
3020 RETURN
```

This program moves the ?A(I) statement out of the subroutine and puts it into a nested subroutine. Nothing else changes.

Computed GOSUB Statement

GOTO and GOSUB statement logic is very similar. The only difference is that GOSUB remembers the next line number. It will therefore not come as any surprise that **there is a computed GOSUB statement akin to the computed GOTO statement.** The computed GOSUB statement allows you to branch to one of two or more subroutines depending on the value of an index. Consider the following statement:

```
90
100 ON A GOSUB 1000,500,5000,2300
110
```

When the statement on line 100 is executed, if A=1 the subroutine beginning at line 1000 is called. If A=2 the subroutine beginning at line 500 is called. If A=3 the subroutine beginning at line 5000 is called. If A=4 the subroutine beginning at line 2300 is called. If A has any value other than 1, 2, 3, or 4, an error message will be reported and the program will stop executing. The computed GOSUB statement remembers the next line number (in this case 110). It does not matter which of the subroutines gets called, the called subroutine's RETURN statement will cause a branch back to the "remembered" line number, in this case line 110.

You can nest subroutines using computed GOSUB statements, just as you can nest subroutines using standard GOSUB statements.

IF-THEN Statement

The arithmetic and relational operators which we described earlier in this chapter are frequently used in **IF-THEN** statements. This **gives a BASIC program decision-making capabilities. Following IF you enter any expression. If the expression is "true," then the statement(s) following THEN are executed. However if the expression is "false" the statement(s) following THEN are not executed.** Here are three simple examples of IF-THEN statements:

```
10 IF A=B+5 THEN PRINT MSG1
40 IF CC$<"M" THEN IN=0
50 IF Q<14 AND M<>M1 GOTO 66
```

The word THEN is optional; it may be omitted, as in the third example.

The statement on line 10 causes a PRINT statement to be executed if the floating point variable A value is five more than the floating point variable B value. The PRINT statement will not be executed otherwise.

The statement on line 40 sets floating point variable IN to 0 if string variable CC$ is any letter of the alphabet in the range A through L.

The statement on line 50 causes program execution to branch to line 66 if floating point variable Q is less than 14, and floating point variable M is not equal to floating point variable M1. Otherwise program execution will continue with the statement on the next line.

If you do not understand the evaluation of expressions following IF, then refer to the discussion of such expressions given at the beginning of this chapter.

INPUT AND OUTPUT STATEMENTS

From the beginning of this chapter we have been using the question mark (?) to create displays. In fact the question mark is a shorthand version of the PRINT statement.

There are a variety of **BASIC statements that control the transfer of data to and from the computer.** Collectively these **are referred to as input/output statements. The simplest input/output statements control data input from the keyboard and data output to the display.** We are going to discuss these simple input/output statements in the paragraphs that follow. But there are also more complex input/output statements that control data transfer between the computer and peripheral devices such as cassette units, diskette units, and printers. These more complex input/output statements are described in Chapter 6.

Since we have already encountered the PRINT statement, let us discuss this statement first.

PRINT Statement

You can use the word PRINT or a question mark (?) to create a PRINT statement.

Why use PRINT instead of DISPLAY or some abbreviation of the word display? The answer is that in the early sixties, when the BASIC programming language was being created, displays were very expensive and generally unavailable on medium- or low-cost computers. The standard computer terminal had a keyboard and a printer. Information was printed where today it is displayed; hence the use of the word "print" to describe a statement which causes a display.

The PRINT statement will display text or numbers. Text must be enclosed in quotes. For example, the following statement will display the single word "text":

```
10 PRINT "TEXT"
```

or:

```
10 ?"TEXT"
```

To display a number, you place the number, or a variable name, after PRINT. This may be illustrated as follows:

```
10 A%=10
20 ?5,A%
```

The statement at line 20 displays the number 5, and then the number 10 on the same line.

You can display a mixture of text and/or numbers by listing the information to be displayed after PRINT. Use commas to separate individual items. The following PRINT statement displays the words "one," "two," "three," "four" and "five," followed by the numeral for each number:

```
10 ?"ONE",1,"TWO",2,"THREE",3,"FOUR",4,"FIVE",5
```

If you separate variables with commas, as we did above, **then the CBM computer automatically assigns 10 character spaces for each variable displayed.** Try executing the statement illustrated above in immediate mode to prove this to yourself. **If you want the display to take out empty spaces, separate the variables with semicolons,** as follows:

```
10 PRINT "ONE";1;"TWO";2;"THREE";3;"FOUR";4;"FIVE";5
```

Enter this statement in immediate mode and display it to understand how the semicolon works.

A PRINT statement automatically inserts a carriage return at the end of the display, unless you suppress it. You can suppress the carriage return by putting a comma or a semicolon after the last variable. A comma occurring after the last variable will continue the display at the next 10-character space boundary. To illustrate this, enter the following three-statement program and run it by typing in RUN:

```
10 PRINT "ONE",1,"TWO",2
20 PRINT "THREE",3,"FOUR",4
30 GOTO 30
```

Now add a comma to the end of the statement on line 10 and again execute the program by typing RUN. You will see the two lines of display occur on a single line. Remember to type RUN on a blank line or you will get a syntax error.

Now replace the comma at the end of line 10 with a semicolon and again run the program. The display occurs on a single line, but the space between the numeral "2" and the word "three" has been removed. By changing other commas to semicolons you can selectively remove additional spaces.

We have been illustrating the numerals by inserting them directly into the PRINT statement. You can, if you wish, display the contents of variables instead. The following program reproduces the first PRINT statement, but uses variable A%(I) to create digits. Try entering this program and running it:

```
10 FOR I=1 TO 5
20 A%(I)=I
30 NEXT
40 PRINT "ONE";A%(1);"TWO";A%(2);"THREE";A%(3);"FOUR";A%(4);
   "FIVE";A%(5)
50 GOTO 50
```

We can put the displayed words into a string array and move the PRINT statement into the FOR-NEXT loop by changing the program as follows:

```
10 DATA "ONE","TWO";"THREE","FOUR","FIVE"
20 FOR I=1 TO 5
30 A%(I)=I
40 READ N$(I)
50 PRINT N$(I);A%(I);
60 NEXT
70 GOTO 70
```

The program shown above is not well written. A%(I) can be eliminated, and N$ need not be an array variable. Can you rewrite the program using N$ and removing A$(I) entirely?

PRINT Formatting Functions

We use the word "formatting" to describe the process of arranging information on a display (or a printout) so that the information is easier to understand, or more pleasing to the eye. Given the PRINT statement and nothing else, formatting could become a complex and painful chore. For example, suppose you want to display a heading in the middle of the line at the top of the display. Does that mean displaying space codes until you reach the first heading character position? Not only would that be tedious and error prone, it would also waste a lot of memory, since each space code must be converted into an appropriate computer instruction. Fortunately, **CBM BASIC provides three PRINT formatting aides: the SPC, TAB, and POS functions.**

SPC Function

SPC is a space over function. You include SPC as one of the terms in a PRINT statement; after the letters SPC you must include (in parentheses) the number of character positions that you wish to space over. For example, we could display a heading beginning at the left-most character position of the display as follows:

```
10 ?"HEADING"
```

But to center the heading on a 40-column screen display you would first space over 16 character positions as follows:

```
10 ?SPC(16);"HEADING"
```

Notice the semicolon after the SPC function. A comma after SPC will start displaying text at the next 10-character boundary following the number of spaces specified by SPC.

Any time you include the SPC function in a PRINT statement you simply cause the next printed or displayed character to be moved over by the number of positions specified after SPC; no other PRINT statement syntax is changed.

TAB Function

TAB is a tabbing function similiar to typewriter tabbing.

Suppose you want to print or display information in columns. You must first calculate the character position of the line where each column is to begin. This may be illustrated as follows:

```
COLUMN NUMBER
↓
0                 16               32           48
JONES, P. J       431-25-6277      1420.00      258.74
BURKE, P. L       447-71-7614      2025.00      467.64
ROBINSON, L. W    231-80-8421      2150.00      477.04
      etc.             etc.          etc.         etc.
```

In the illustration above, columns begin at character positions 0, 16, 32 and 48. (Obviously the computer has an 80-column display or is printing on 80-column paper.) Now instead of computing space codes as you go from line to line, following each column entry you simply insert a TAB function in the PRINT statement.

Consider one line of the display illustrated above; counting character positions, we could display the line without tab stops, as follows:

```
10 ?"JONES,P.J                431-25-6277             1420.00             258.74"
```

Instead of inserting space codes, we could use the space function and shorten the statement as follows:

```
10 ?"JONES, P.J";SPC(17);"431-25-6277";SPC(5);"1420.00";SPC(9);"258.74"
```

But tabbing is easier because you tab to a known column number instead of counting spaces:

```
10 ?"JONES,P.J";TAB(16);"431-25-6277";TAB(32);"1420.00";TAB(48);"258.74"
```

Note that the entries in the third and fourth columns are numbers which we have entered as text. Try rewriting the PRINT statement to display these as numbers. The numbers no longer align as they did when they were displayed as characters (in Chapter 5 we discuss the quirks associated with display formatting). In this case, numbers leave a space for a negative sign, and they do not display zeros occurring after the decimal point. That is why there are differences.

POS Function

POS is the last of the PRINT formatting functions. POS returns the current cursor position. The position is returned as a number, equal to the column number where the cursor is blinking. You always include a dummy argument of 0 after POS, written as POS(0).

The following statement demonstrates the capability of POS:

```
10 ?"CURSOR POSITION IS";POS(0)
```

Execute this statement in immediate mode. The display will appear as follows:

```
?"CURSOR POSITION IS";POS(0)
CURSOR POSITION IS 18
```

The cursor was at character position 18 after displaying "CURSOR POSITION IS." If you add some spaces after "IS," and before the closing quotes, you will change the number 18 to some larger number.

INPUT Statement

When an INPUT statement is executed, the computer waits for input from the keyboard; until the computer gets the input it requires, nothing else will happen.

An input statement begins with the word INPUT, which is followed by a list of variable names. Entered data is assigned to the named variables. The variable name type determines the form in which data must be entered. A string variable name (ending with a $) can be satisfied only by text input; any number of text characters can be entered for a string variable. To demonstrate string input, key in the following short program and run it:

```
10 INPUT A$
20 ?A$
30 GOTO 10
```

Upon executing an INPUT statement, the computer displays a question mark, then waits for your entry. The program illustrated above displays any text which you enter, as you enter it; but the text is displayed again because of the PRINT statement on the next line. The first display occurs when the INPUT statement on line 10 is executed. The second display is in response to the PRINT statement on line 20.

You input integer or floating point numeric data by listing the appropriate variable names following INPUT. Separate individual entries with commas. The comma has no punctuation significance in an INPUT statement. The following example inputs a text word, an integer number and a floating point number, then displays these three inputs. Enter the program and run it:

```
10 INPUT A$,A,A%
20 ?A$,A,A%
30 GOTO 10
```

You must enter a text word followed by a comma, then an integer number followed by a comma, then a floating point number followed by a carriage return. Any departure from this input sequence will cause an error; following an error the computer displays two question marks. You will have to re-enter the data in the correct format. If the computer then displays a question mark with the message "re-do from start," enter the correct data again.

Now rewrite the PRINT statement so that A$, A and A% are in an order that differs from the INPUT statement. Rerun the program.

As we discussed earlier, any integers can be represented using a floating point number. Therefore you can input an integer value for a floating point variable. But you cannot input a floating point value for an integer variable. You cannot enter text for an integer or a floating point number, but you can enter a number for a text variable; the number will be interpreted as characters rather than a numeric value. Try these variations to satisfy yourself that you understand the data entry options.

The INPUT statement is very fussy; its syntax is too demanding for any normal human operator. Just imagine the office worker who knows nothing about programming; on encountering the types of error message which can occur if one comma happens to be out of place, s/he will give up in despair. You are therefore likely to spend a lot of time writing "idiot-proof" data entry programs; these are programs which are designed to watch out for every type of mistake that an operator can make when entering data. An idiot-proof program will cope with errors in a way that the operator can understand. Chapter 5 describes data entry programming in detail.

One simple trick worth noting, however, is the INPUT statement's ability to display data. Therefore you can precede each item of data entry with a short message telling the operator what to do. The message appears in the INPUT statement as text between quotes. A semicolon must occur after the text to be displayed, and before the first input variable name. Here is an example:

```
10 INPUT "ENTER THE NUMBER 1";N
20 IF N<>1 THEN GOTO 50
30 ?"OK"
40 GOTO 40
50 ?"NO, DUMMY,"
60 GOTO 10
```

This program prints a message, then waits for a single data entry. This certainly beats sticking a bunch of variables into a single INPUT statement, with only your memory reminding you what to enter next.

GET Statement

The GET statement inputs a single character. No carriage return is needed. The single character input can be any character that the CBM computer recognizes, or it may be a numeric value between 0 and 9. Entry will be interpreted as a character if a string variable name follows GET. Type in the following program and run it:

```
10 GET A$
20 ?A$
30 GOTO 10
```

When you run this program, everything will race off the top of the display. Each time you press a key, it too will race off the top of the screen. That is because GET does not wait for a character entry, it assumes the entry is there. **We can make GET wait for a specific character** by testing for the character as follows:

```
10 GET A$
20 IF A$<>"X" THEN GOTO 10
30 ?A$
40 GOTO 10
```

This program waits for the letter *X* to be entered. Nothing else will do.

GET can also be programmed to wait for any keyboard entry. This program logic uses the fact that the GET statement string variable is assigned a null character code until a character is input at the keyboard. The null code is 00 which cannot be entered from the keyboard, but can be specified within a program, using two adjacent quotation marks "". Here is the necessary program logic:

```
10 GET A$
20 IF A$="" THEN GOTO 10
30 ?A$
40 GOTO 10
```

If the GET statement specifies an integer or floating point variable, then the input is interpreted as a numeric digit. The integer of floating point variable appearing in a GET statement is assigned a value of 0 until it receives data input. But you can enter 0 at the keyboard. Therefore **program logic has no way of knowing whether the 0 represents a valid entry, or a lack of any entry.** This can present problems to programming logic that checks for an entry, as shown above. **GET statements therefore usually receive string characters.**

Programs use the GET statement most frequently when generating dialogue with an operator. For example, a program may wait for an operator to prove that he or she is there by entering a specific character (e.g. 'Y' for 'yes'). Here is appropriate program logic:

```
10 PRINT "OPERATOR! ARE YOU THERE? TYPE Y FOR YES"
20 GET A$
30 IF A$<>"Y" THEN GOTO 20
40 PRINT "OK, LET'S GET ON WITH IT"
```

Notice that this sequence never displays the character entered at the keyboard. Try rewriting the program so that any character entered for the GET statement is displayed.

PEEK AND POKE STATEMENTS

PEEK and POKE are two CBM BASIC statements that rightfully belong in Chapter 7; however we will mention them here since we have already encountered the POKE statement in the course of operating the CBM computer. We used it to access the computer's alternate character set.

CBM computers can have up to 65,536 individually addressable locations, each of which can store a number ranging between 0 and 255. (This strange upper bound is in fact 2^8-1.) All programs and data are converted into sequences of numbers which are stored in this fashion.

A PEEK statement lets you read the number stored in any CBM computer memory location. Consider the following PEEK statement:

```
10 A%=PEEK(200)
```

This statement assigns the content of memory location 200 to variable A%. The PEEK argument may be a number, as shown, an integer variable name, or an integer expression, but it must evaluate to the address of a memory location.

The POKE statement writes data into a memory location. For example the statement:

```
20 POKE 8000,A%
```

takes the content of variable A% and stores it in memory location 8000. Each POKE argument may be a number, a variable or an expression with a value between 0 and 255. A floating point value is converted to an integer.

You can PEEK into read/write memory or read-only memory. But you can only POKE into read/write memory. This is self-evident; read-only memory, as its name implies, can have its contents read, but cannot be written into.

END AND STOP STATEMENTS

The END and STOP statements halt program execution. You can continue execution by typing CONT at the keyboard. You do not have to include END or STOP statements in your program; however these statements do make for tidy programming.

In many of the programming examples given in this chapter we use a GOTO statement that branches to itself in order to stop program execution. For example the statement:

```
50 GOTO 50
```

will execute endlessly since the GOTO statement selects itself for the execution. We could replace this statement with a STOP statement. When a STOP statement is executed, the following message will appear:

```
BREAK IN XXXX
READY
```

Then execution stops. XXXX is the line number of the STOP statement. If you have more than one STOP statement in your program, use XXXX to identify which statement was executed.

FUNCTIONS

Another element of CBM BASIC is the function, which in some ways looks like a variable, but in other ways acts more like a BASIC statement.

Perhaps the simplest way of understanding what a function is is to look at an example in an assignment statement:

```
10 A=SQR(B)
```

The variable A has been set equal to the square root of the variable B. SQR specifies the square root function. Here is a string function:

```
20 C$=LEFT$(D$,2)
```

In this example the string variable C$ is set equal to the first two characters of string variable D$.

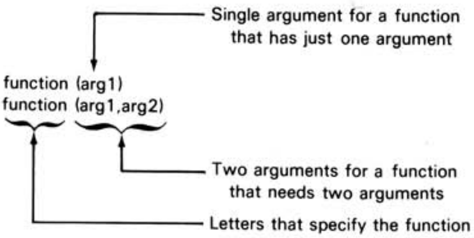
Functions can substitute for variables or constants anywhere in a BASIC statement, except to the left of an equal sign. In other words, you can say that A=SQR(B), but you cannot say that SQR(A)=B.

We have already used four functions. **SPC, TAB, and POS are system functions used with the PRINT statements to format displays. Also, PEEK is a function.**

The discussion which follows shows you how to use functions. An incomplete summary of the available CBM BASIC functions is presented here but **complete descriptions of all functions are given in Chapter 8.**

You specify a function using appropriate letters (such as SQR for square root), followed by arguments enclosed in parentheses. In the case of A=SQR(B), SQR requires a single argument, which in this case is the variable B. For C$=LEFT$(D$,2), LEFT$ specifies the function; the two arguments D$ and 2 are enclosed in brackets.

Generally stated, any function will have one of these two formats:


Single argument for a function that has just one argument
function (arg1)
function (arg1,arg2)
Two arguments for a function that needs two arguments
Letters that specify the function

A few functions need three arguments.

Each function argument can be a constant, a variable, or an expression.

A function appearing in a BASIC statement is evaluated before any operators. Each and every function in a BASIC statement is reduced to a single numeric or string value before any other parts of the BASIC statement are evaluated. For example in the following statement:

```
10 B=24.7*(SQR(C)+5)-SIN(0.2+D)
```

SQR and SIN functions are evaluated first. Suppose SQR(C)=6.72 and SIN(0.2+D)=0.625. The statement on line 10 will first be reduced to:

```
10 B=24.7*(6.72+5)-0.625
```

Then this simpler statement is evaluated.

ARITHMETIC FUNCTIONS

Here is a list of the arithmetic functions that you can use with CBM BASIC:

INT	Converts a floating point argument to its integer equivalent by truncation.
SGN	Returns the sign of an argument: +1 for a positive argument, −1 for a negative argument, 0 for 0 argument.
ABS	Returns the absolute value of an argument. A positive argument does not change; a negative argument is converted to its positive equivalent.
SQR	Computes the square root of the argument.
EXP	Raises the natural logarithm base e to the power of the argument (e^{arg}).
LOG	Returns the natural logarithm of the argument.
RND	Generates a random number. There are some rules regarding use of RND; they are described in Chapter 5.
SIN	Returns the trigonometric sine of the argument, which is treated as a radian quantity.
COS	Returns the trigonometric cosine of the argument, which is treated as a radian quantity.
TAN	Returns the trigonometric tangent of the argument, which is treated as a radian quantity.
ATN	Returns the trigonometric arctangent of the argument, which is treated as a radian quantity.

You should start using functions as soon as possible, but do not bother with functions you do not already understand. For example, if you do not understand trigonometry, you are unlikely to use SIN, COS and TAN functions in your programs.

Here is an example that uses an arithmetic function:

```
10 A=2.743
20 B=INT(A)+7
30 ?B
40 STOP
```

When you execute this program, the result displayed is 9, since the integer value of A is 2. As an exercise, change the statement on line 10 to an INPUT. Change line 40 to GOTO 10. Now you can enter a variety of values for A and watch the integer function at work.

Here is a more complex example using arithmetic functions:

```
10 INPUT A,B
20 IF LOG(A)<0 THEN A=1/A
30 ?SQR(A)*EXP(B)
40 GOTO 10
```

If you understand logarithms, then as an exercise change the statement on line 20, replacing the LOG function with arithmetic functions that perform the same operation.

The argument of a function can be an expression; the expression may contain functions. For example, change line 30 to the following statement and rerun the program:

```
30 ?SQR(A*EXP(B)+3)
```

Now experiment with arithmetic functions by creating immediate PRINT statements that make complex use of arithmetic functions.

STRING FUNCTIONS

String functions allow you to manipulate string data in a variety of ways. You may not need to use arithmetic functions that you do not understand, but you must make the effort to learn every string function.

Here is a list of the string functions that you can use with CBM BASIC:

STR$	Converts a number to its equivalent string of text characters.
VAL	Converts a string of text characters to their equivalent number (if such a conversion is possible).
CHR$	Converts an 8-bit binary code to its equivalent ASCII character.
ASC	Converts an ASCII character to its 8-bit binary equivalent.
LEN	Returns the number of characters contained in a text string.
LEFT$	Extracts the left part of a text string. Function arguments identify the string and its left part.
RIGHT$	Extracts the right part of a text string. Function arguments identify the string and its right part.
MID$	Extracts the middle section of a text string. Function arguments identify the string and the required mid part.

String functions let you determine the length of a string, extract portions of a string, and convert between numeric, ASCII, and string characters. These functions take one, two, or three arguments. Here are some examples:

```
STR$(14)

LEN("ABC")

LEN(A$+B$)

LEFT$(ST$,1)
```

SYSTEM FUNCTIONS

In the interest of completeness, CBM BASIC system functions are listed below. They perform operations which you are unlikely to need until you are an experienced programmer. Perhaps the only system function you are likely to use fairly soon is the time of day function. If you print many variations of a report (or any other material) in a single day, it is often a good idea to print the time of day at the top of the report. Then you can tell the sequence in which these reports were generated.

Here is a list of system functions available with CBM BASIC:

PEEK	Fetches the contents of a memory byte.
TI$, T1	Fetches system time, as maintained by a program clock.
FRE	Returns available free space — the number of unused read/write memory bytes.
SYS	Transfers to subsystem.
USR	Transfers to user assembly language program.

USER-DEFINED FUNCTIONS

In addition to the many functions which are a standard part of CBM BASIC, **you can define your own arithmetic functions,** providing they are not very complicated. User-defined string functions are not allowed. Here is an example of a short program that uses a DEF FN statement:

```
10 DEFFNP(X)=100*X
20 INPUT A
30 ?A,FNP(A)
40 GOTO 20
```

Following the DEF FN entry you can have any valid floating point variable name. In this case we have entered P, therefore the function name becomes FNP. If the variable name was AB, then the function name would be FNAB.

In a DEF FN statement, a single variable name must follow the function name, and must be enclosed in parentheses. This variable name is local to the function definition; its value is known only inside the DEF FN statement. You can use the same variable name outside the function, but it refers to a different variable value which is known to the program at large. The local variable receives its value when the function can, and usually does, appear in the expression on the right side of the DEF FN statement equals sign. Other variable names can appear there too. When the function is used via the FN statement, the expression is evaluated using the newly assigned value of the local variable and the latest values of any of the variables. The resulting value is used where the FN statement appeared.