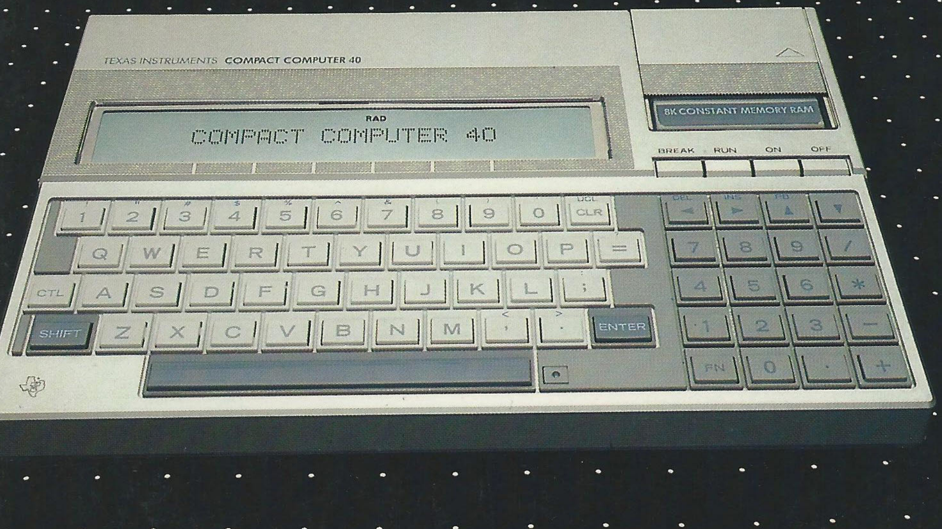
A BYTE BOOK

>>$9.95

LEARN BASIC: A Guide to Programming the Texas Instruments Compact Computer 40

DAVID THOMAS



LEARN BASIC

LEARN BASIC

A Guide to Programming the Texas Instruments Compact Computer 40

David Thomas

with the assistance of Texas Instruments

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland Bogotá
Guatemala Hamburg Johannesburg Lisbon London Madrid
Mexico Montreal New Delhi Panama Paris San Juan
São Paulo Singapore Sydney Tokyo Toronto

LEARN BASIC: A Guide to Programming the Texas Instruments Compact Computer 40

McGraw-Hill/VTX Series. A BYTE Book. McGraw-Hill edition.

1 2 3 4 5 6 7 8 9 0 S E M S E M 8 9 8 7 6 5 4 3

ISBN 0-07-064257-5

LIBRARY OF CONGRESS CATALOGING IN PUBLICATION DATA

Thomas, David, date
Learn BASIC.

Includes index.
1. TI CC 40 (Computer)—Programming. 2. Basic (Computer program language) I. Title.
QA76.8.T136T48 1983 001.64'24 82-24905
ISBN 0-07-064257-5

The editors for this book were John A. Aliano and Jeffrey McCartney; and the editing supervisor was Charles P. Ray.
Book design by Roberta Rezk.

Contents

Preface

The objective of this book is to help you learn BASIC and to sho you how to use that language to program the Texas Instrumen Compact Computer 40. This book assumes that you have the *Con pact Computer 40 User's Guide* supplied with the CC-40 and th you have read the discussions regarding the operation of the con puter and the functions of the keys. The book does not assum that you understand everything in that manual, or that you hav had previous computer programming experience. On the contrar it assumes that you are now making your first efforts to learn ho to program a computer. If you have had previous programming e perience, this book will help you utilize the full capabilities of th CC-40.

There are 28 lessons in this book. Each lesson covers an importa programming concept or BASIC statement or command and is sho enough to be easily read in one session. The lessons gradual build on the information discussed in the preceding lessons, s read the lessons in sequence. At the end of each lesson is a sun mary section. Use this section for a quick review and reinforceme of the contents of the lesson.

The only way to learn to program is to write programs and to ru them. This book makes that easy. It contains example program to illustrate every BASIC statement and command discussed. Th example programs, however, are designed to do more than ju show how a statement or command works. They also are intended to arouse your interest and provide you with a core of useful programming applications that you can adapt or expand to suit your own needs. After reading the lessons and entering the given programs, try to write alternate versions. There are few things as flexible as programming, and there are many ways of making the computer perform the same actions. By composing programs of your own design, you will more quickly develop your programming skills.

You will find a review test after every fourth or fifth lesson. The questions in these tests have been selected to reinforce what you have learned in the preceding lessons and challenge you to write programs using that information. Take the time to work these exercises. Complete answers to all questions are provided in Appendix C at the back of the book. The explanations accompanying many of the answers make additional points that were not specifically detailed in the text.

One of the remarkable things about computer programming is that you can learn to program from reading a book. The only requirements are enthusiasm, a desire to learn, and your own computer. As you learn the BASIC programming language, one of today's most useful tools, I trust that you will discover, as I have, that there are few activities more fascinating or rewarding than computer programming.

David Thomas

Acknowledgments

The author wishes to express his gratitude to Mark R. Waldin, whose idea made this book possible; to Robert E. Whitsitt II for the support needed to write this book; and to Tom M. Ferrio and Floyd R. Gerwig for their patient reading and many valuable suggestions and criticisms.

For their help in proofreading the manuscript, I wish to thank Frank Dever, Jeff Gray, Millie Hammond, Kenneth Edward Heichelheim, Ben Korte, and Walter L. Wdowiak.

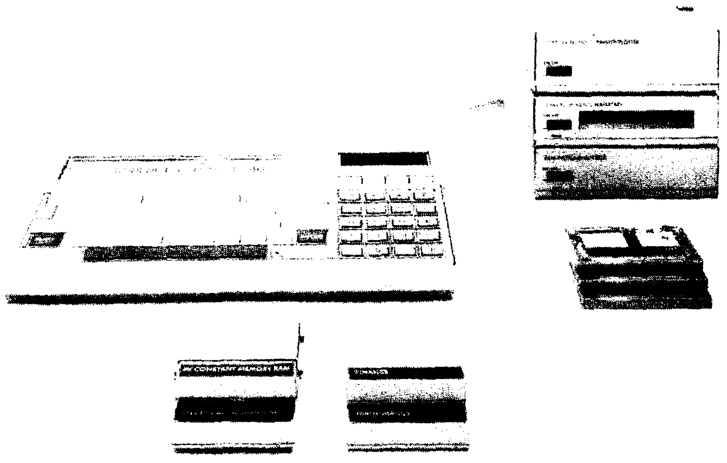


Fig. 1-1 Texas Instruments Compact Computer 40 and peripheral devices; (right background, top to bottom): TI four-color Printer/Plotter, TI Wafertape™ Digital Tape Drive, and TI RS232 Communication Interface; (right foreground): Wafertape™ digital tape cassettes; (center foreground, left to right, top to bottom): cartridges for 8K Constant Memory™ RAM and Solid State Software™ programs in electrical engineering, finance, and mathematics.

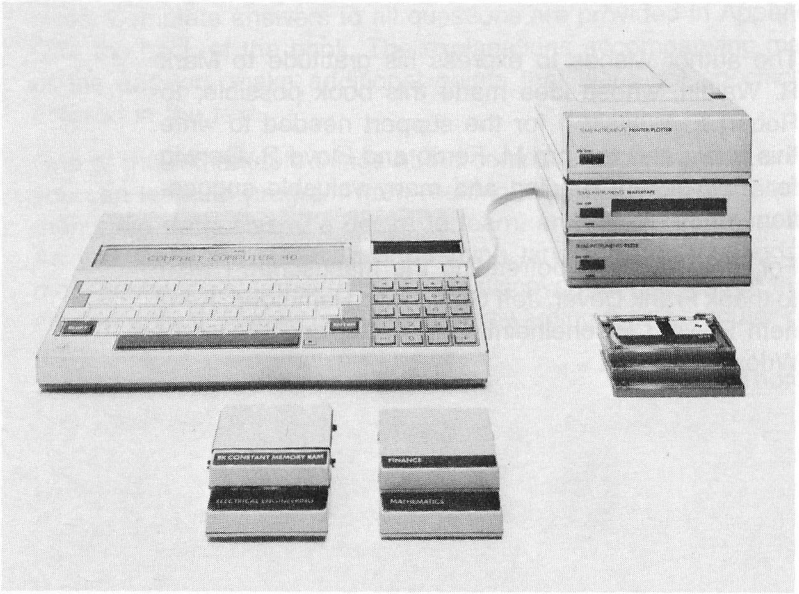


Fig. 1-1 Texas Instruments Compact Computer 40 and peripheral devices; (right background, top to bottom): TI four-color Printer/Plotter, TI Wafertape™ Digital Tape Drive, and TI RS232 Communication Interface; (right foreground): Wafertape™ digital tape cassettes; (center foreground, left to right, top to bottom): cartridges for 8K Constant Memory™ RAM and Solid State Software™ programs in electrical engineering, finance, and mathematics.

Lesson 1

The TI Compact Computer 40

The Texas Instruments Compact Computer 40 is a remarkably powerful computing system. The textbook size of the computer makes it ideal for "on-the-spot" applications that require immediate solutions anywhere you go. Its usefulness, though, extends beyond its portability. With its powerful BASIC programming language, expandability, and calculating accuracy, the CC-40 can do many things that previously could be done only by large, immobile "main-frame" computers. (See Fig. 1-1.)

The CC-40 is equipped with 6144 bytes of random access memory (6K RAM). This is enough memory and power to write sophisticated programs, work complex business and science problems, or store several pages of data. If more memory is needed, the computer can be expanded to as much as 22,528 bytes of memory by installing a 16K RAM expansion cartridge in the computer's cartridge port. The cartridge port can also be used to run TI Solid State Software™ preprogrammed application cartridges.

Directions to the computer and information to be stored in memory are entered by means of a standard typewriter-like keyboard. Although the compact size of the computer results in keys that are smaller than those found on a normal typewriter, with a little practice, it is still possible to touch type. To the right of the main keyboard is a block of keys known as a "numeric keypad." Besides duplicating the number keys on the top row of the keyboard, the numeric keypad

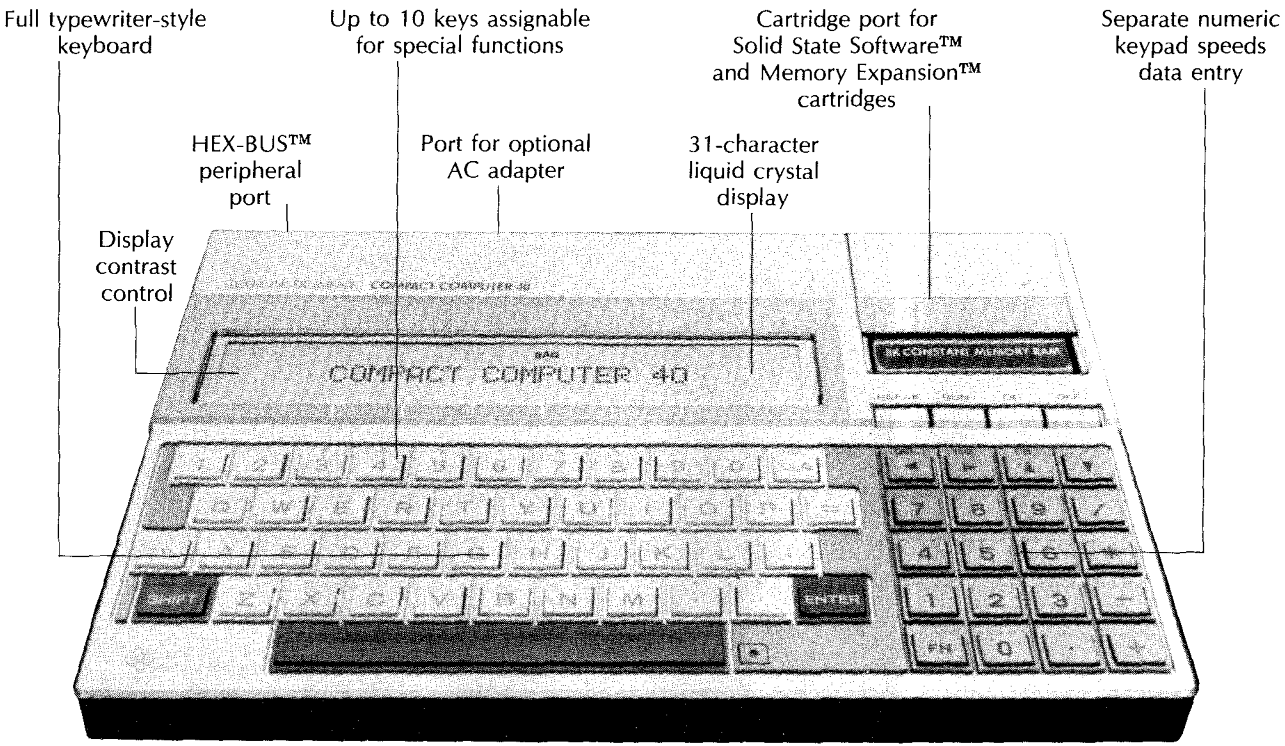

Full typewriter-style keyboard
Up to 10 keys assignable for special functions
Cartridge port for Solid State Software™ and Memory Expansion™ cartridges
Separate numeric keypad speeds data entry
HEX-BUS™ peripheral port
Port for optional AC adapter
31-character liquid crystal display
Display contrast control
COMPACT COMPUTER 40
SHIFT
ENTER

Fig. 1-2 The Texas Instruments CC-40 standard QWERTY keyboard and numeric keypad with cursor control keys make calculations and text and data entry quick and easy.

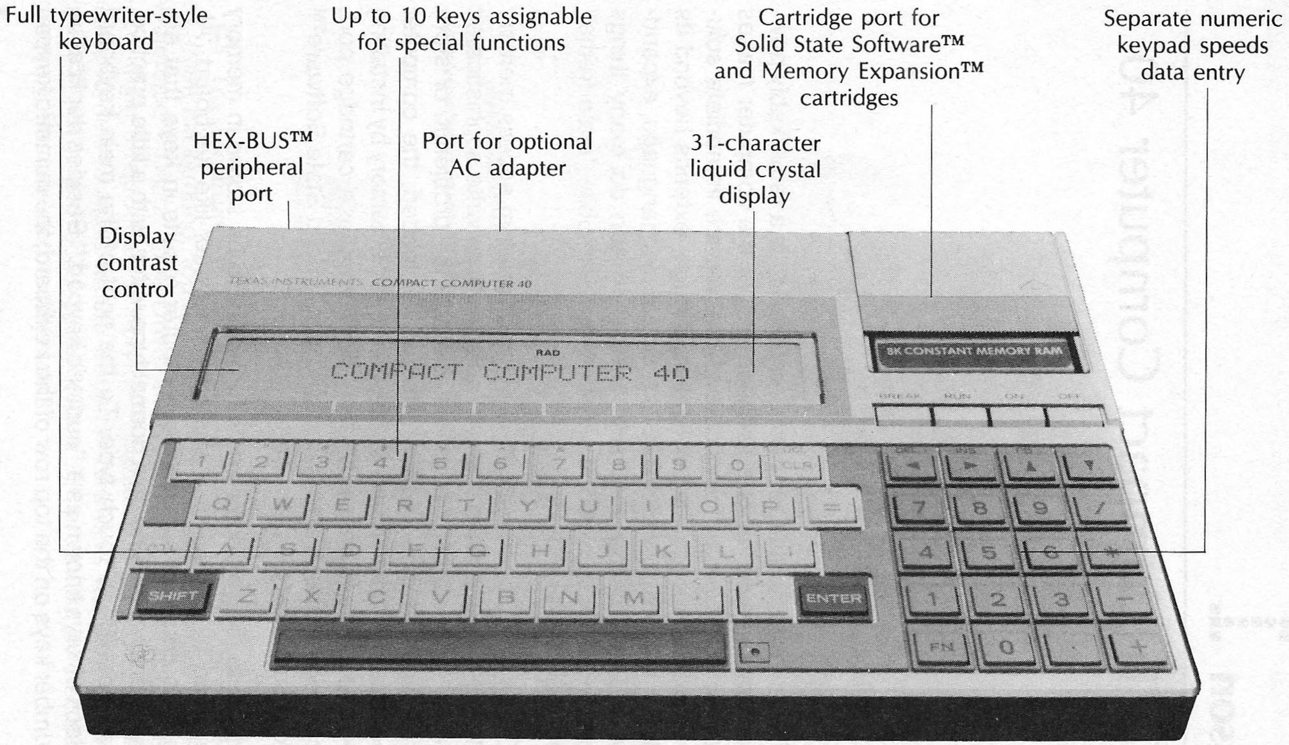

Full typewriter-style keyboard
Up to 10 keys assignable for special functions
Cartridge port for Solid State Software™ and Memory Expansion™ cartridges
Separate numeric keypad speeds data entry
HEX-BUS™ peripheral port
Port for optional AC adapter
31-character liquid crystal display
Display contrast control
TEXAS INSTRUMENTS COMPACT COMPUTER 40
RAD
COMPACT COMPUTER 40
8K CONSTANT MEMORY RAM
BREAK
RUN
ON
OFF
SHIFT
ENTER
FN

Fig. 1-2 The Texas Instruments CC-40 standard QWERTY keyboard and numeric keypad with cursor control keys make calculations and text and data entry quick and easy.

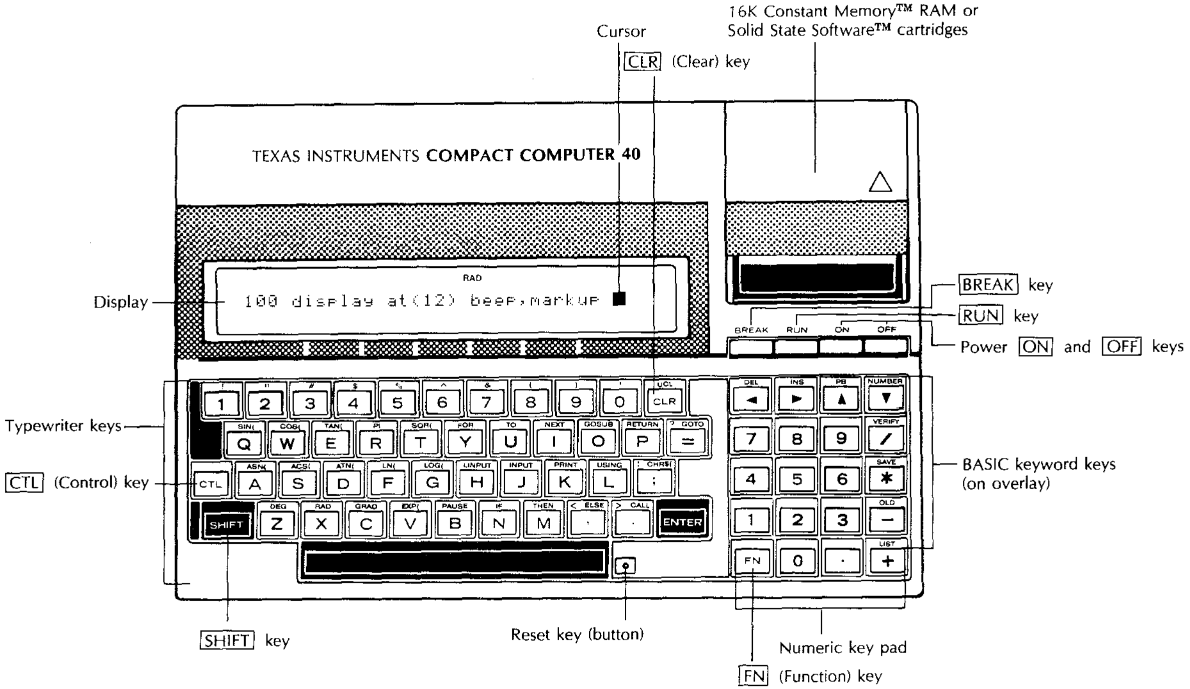


Fig. 1-3 The dot-matrix design of the Texas Instruments CC-40's liquid crystal display allows display of a full range of both uppercase and lowercase letters, punctuation marks, signs, and symbols.

contains editing keys and the mathematical calculation keys of addition, subtraction, multiplication, and division. The numeric keypad permits calculations and data entry to be performed more quickly than is possible with a typewriter keyboard. (See Fig. 1-2.)

The results of programs and calculations, as well as the contents of memory, are viewed through the 31-character liquid crystal display built into the computer. (See Figs. 1-3 and 28-1.) The dot-matrix design allows you to display both uppercase and lowercase letters, as well as various punctuation marks, signs, and symbols. You can even define your own symbols by specifying which dots the computer is to turn on.

The computer is powered by four AA-size batteries that maintain the contents of memory even when the computer is turned off. The advantage of this feature is that programs and data can be conveniently stored in the computer without the fear of loss. The power consumption of the computer is low enough to permit up to 200 hours of continuous operation before new batteries are needed. This allows several hours of daily use for up to 6 months between battery changes. Additionally, the computer is equipped with a port for an optional AC adapter.

The heart of the computer is a new Texas Instruments 8-bit microcomputer known as the TMS70C20. The microcomputer performs the operations entered through the keyboard or specified in a program and places the results in the display. If requested, it also sends and receives information from other devices such as the Wafertape™ Digital Tape Drive or the RS232 Communications Interface. (See Fig. 1-1.) These devices, known as peripherals, are connected to the Compact Computer 40 through the HEX-BUS™ Intelligent Peripheral Interface located in the rear of the computer. Each peripheral has its own computing power provided by a built-in dedicated microcomputer.

The availability of such powerful peripherals as the Wafertape™ Digital Tape Drive, TI RS232 Communications Interface, and TI Printer/Plotter expand the usefulness of the computer enormously. The Wafertape™ Drive allows you to store programs and data on inexpensive, endless-loop tape cartridges. These tape cartridges are available in several storage capacities, up to a maximum of 49,152 bytes (48K). Since the tapes never require rewinding, they permit all storage operations to be completely controlled by the computer. The Wafertape™ Drive unit is powered by four AA batteries operating independently of computer console power, and is fully portable.

The RS232 Communication Interface allows the CC-40 to be connected to a whole range of standard computer equipment, including devices which allow the CC-40 to communicate with other appropriately programmed computers throughout the world over regular telephone lines. This allows you to use your CC-40 as a remote terminal from your home, office, or even a motel room. The RS232 Interface can be purchased with an optional parallel printer port that allows you to print programs and data on most 80-column printers.

The Printer/Plotter peripheral can draw four-color X-Y graphs and make permanent hard-copy printed records of your programs, data, and solutions. With a printing speed of up to 11 characters per second, the Printer/Plotter can print in 10 different type sizes and even permits words to be printed upside down or vertically, if desired. The Printer/Plotter uses standard 2¼-inch plain paper rolls and prints in red, blue, green, and black. Power for the Printer/Plotter is supplied from rechargeable AA batteries.

The Texas Instruments Compact Computer 40, with its advanced calculation capability, enhanced built-in BASIC language, and flexible peripherals, provides you with the ideal opportunity to learn the BASIC programming language. As computers become increasingly important in our daily lives, it is becoming just as important to acquire a familiarity and understanding of computers. Learning the BASIC programming language is the first step in acquiring that proficiency. The following lessons offer a practical, hands-on approach to learning BASIC, one of the most popular computer programming languages.

Lesson 2

Numeric Constants and Variables

The Compact Computer's BASIC language allows all but extremely large or small numbers to be entered exactly as they are written. Decimal numbers are entered with a decimal point, if needed, and negative numbers are entered with a minus sign preceding the number.

To acquaint yourself with the entry of numbers, work the following problems with the computer. The computer does not need an equals key $\boxed{=}$ to perform a calculation; simply type in the problem and press the $\boxed{\text{ENTER}}$ key. To clear the result of a previous calculation from the display, just start typing the next calculation.

Enter	**Display**
121+121	`242`
7.123−9	`-1.877`
629*5.2	`3270.8`
3/17	`.1764705882`

(Note that the $\boxed{*}$ key performs multiplication and the $\boxed{/}$ key performs division.)

Numbers such as those illustrated above are known as numeric *constants,* since their value cannot vary. In programming, you will more often use numeric *variables* to represent numbers. A numeric variable is a "name" assigned a number value that can be used in place of a numeric constant.

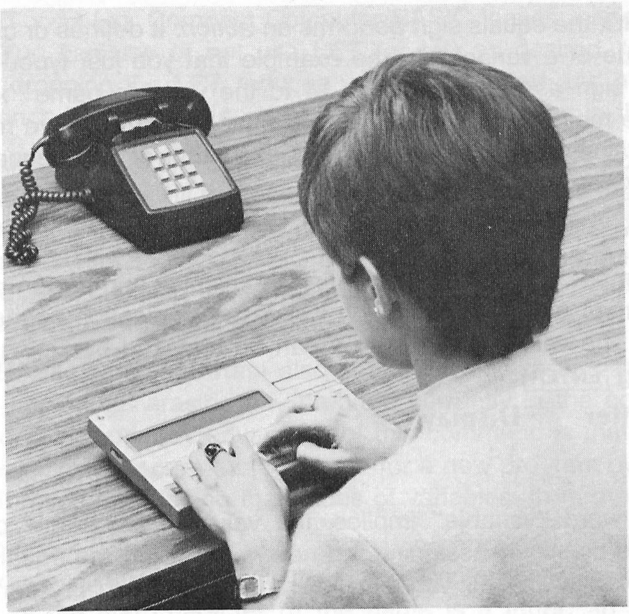
The Texas Instruments CC-40 keyboard design allows easy keyboarding for touch-typists.

Variables are assigned number values in BASIC using the equals ( ) sign. For example, type

```
x=14
```

into the computer. Before pressing $\boxed{\text{ENTER}}$, consider the above sequence carefully. It illustrates an important difference between the use of the equals sign in BASIC and the use of the equals sign in standard mathematics.

In mathematics, the equals sign indicates that two values or expressions are equal. For example, $2 + 3 = 5$ is a familiar mathematical equation indicating that $2 + 3$ and 5 are equal. The equals sign in this usage simply states a fact, that the two sides of the equation are equal.


The Texas Instruments CC-40 keyboard design allows easy keyboarding for touch-typists.

In BASIC, the equals sign performs an *action:* it defines or changes the value of a variable. In the example that you just typed in, the equals sign *assigns* the value 14 to the variable name "x". The variable may have had no previous value, or it may have had any other value. In any case, after the execution of this sequence, x will have a value of 14.

Now press ENTER to execute the sequence.

The computer responds by clearing the display and performing the assignment operation, although it gives no indication of the latter action. The value assigned to x can be checked by typing x and pressing ENTER:

Enter	**Display**
x	14

As the word "variable" implies, the value of a variable can be changed by simply assigning it a new value using the equals sign. This flexibility makes variables extremely useful in programming.

Type

```
x=x+x
```

and press ENTER. Then perform the check again.

Enter	**Display**
x	28

The new value of x is 28. It was obtained by assigning x the value of x + x. Notice that the "old" value of x was used in assigning a new value to x. A BASIC sequence such as x = x + x cannot be confused with a mathematical sequence, since it makes no sense as an equation: it would be like saying 1 = 1 + 1.

When assigning a value to a variable, always place the variable name on the left side of the equals sign and the value on the right side. This is a fundamental rule of BASIC programming.

Technical Note: Some BASICs require that assignment sequences be preceded by the BASIC statement **LET** (for example, **LET** x=14 or **LET** x=x+x). Compact Computer BASIC leaves this choice to you. You can use or not use **LET** (or **let**) as desired. However, since dropping the **LET** reduces your typing effort and makes no difference in the computer's operation, it is not used in this book.

The initial value of any numeric variable is zero. This is a convention of BASIC that you will find useful and convenient. Once the variable has been assigned a nonzero value, the variable retains that value until altered by another BASIC instruction or until the computer is turned $\boxed{\text{OFF}}$. Turning the computer $\boxed{\text{OFF}}$ or allowing the Automatic Power Down™ to occur sets the value of all variables to zero.

The BASIC command **new** also sets the value of all variables to zero, as well as erasing any program currently in the computer's memory. The primary advantage of this command is that it allows you to quickly prepare the computer for a new program by clearing all information, including the values of variables, from memory.

In Compact Computer BASIC, a variable name can be almost any sequence of letters and digits up to 15 characters in length, as long as the first character of the name is a letter, an underline ( ) character, or an at (@) sign. For example,

```
x
@payroll
count12
first_new_date
a7c3
 cost
```

are all valid variable names. However,

```
3x
percentconcentration
test 1
b12?
interest-costs
```

are invalid names because they either begin with a number, are too long, or contain characters that are not valid (in these examples, the space, question mark, or hyphen, respectively).

A list of names that would otherwise be valid, but cannot be used because they represent BASIC program commands and statements, is given in Appendix B. For example, the assignment sequence "**let** = 5" is invalid because **let** is part of the BASIC language of the computer. Unlike some BASIC languages, however, Compact Computer BASIC allows variables to contain BASIC words as part of a variable name. For example, **run** is a BASIC word and therefore cannot be used as a variable, but "prune", which contains **run**, can be used.

LEARNED IN THIS LESSON

Numbers entered into the computer can be of two types: constants and variables. Specific values such as 3.313 are referred to as constants. "Names" assigned to numbers are called variables. Variable names must begin with a letter of the alphabet, the underline (_) character, or the @ symbol, *and cannot be on the reserved word list.* The reserved word list is given in Appendix B of this book.

The equals sign is used to assign values to variables. The variable name must always be on the left side and the value on the right side of the equals sign.

Until assigned another value, all numeric variables are equal to zero. Once assigned a nonzero value, a numeric variable retains that value in memory until it is replaced or cleared. The value of a variable can be replaced by assigning it a new value. It can be cleared by turning the computer $\boxed{\text{OFF}}$ or executing the **new** command. The **new** command instructs the computer to set all numeric variables to zero and to erase any program currently in memory.

Lesson 3

DISPLAYing Information

The CC-40 is equipped with a 31-character liquid crystal display (LCD). This display is your means of observing the entry of information into the computer, monitoring its operation, and viewing the contents of memory and the results of calculations and programs. The computer automatically displays the results of calculations. This greatly increases the convenience of using the computer as a pocket calculator, but is unusual for a BASIC language computer. To see the results of other types of actions, you must use a specific instruction.

The **display** statement is Compact Computer BASIC's primary method of placing information in the display. For example, enter

```
cost=10.25
```

Now display the value of the variable using the **display** statement:

Enter	**Display**
display cost	10.25

The value is displayed with a nonflashing underline in the left side of the display. The underline cursor indicates the computer is in the "pause" state. The purpose of the pause state is discussed in Lesson 4. For the moment, be aware that when you see a nonflashing underline you must press CLR or ENTER before you can proceed with another operation.

The **display** statement is extremely powerful and offers several optional parameters. For example, you can display information at any character position with the **at(*column*)** option. A comma must separate the **at** parameter from the information being displayed.

Enter	**Display**
display at(5),cost	10.25

This form of the **display** statement instructs the computer to display the value beginning at the fifth column position. If you look closely, however, you will observe that the number actually begins in the *sixth* column position. This apparent discrepancy occurs because BASIC places a "character space" in front of a positive number. This space corresponds to the position occupied by the minus sign of a negative number and is added for consistency. The "leading" space in front of a positive number ensures that the digits of both negative and positive numbers begin in the same relative location:

Enter	**Display**
display at(5),−10.25	-10.25

In addition to adding a leading space before positive numbers, BASIC also adds a space after every number. This "trailing" space is intended to separate the number from any information that may be displayed following the number.

The valid range for column positions is 1 through 80, inclusive, but only the first 31 positions correspond to physical locations in the display. For example, try

Enter	**Display**
display at(32),cost	

The underline cursor is displayed as usual, but no value is visible. To see the value, hold down the right arrow ▶ key. The underline cursor moves toward the right side of the display. If you continue to hold the right arrow key down, the value *scrolls* into view 1 character at a time. Since the display is physically limited to 31 characters, it is not possible to see information displayed beyond column 31 unless the information is scrolled into view.

Technical Note: The computer has what is known as an 80-character display buffer. Since the display is only 31 characters wide, it is impossible to see the entire buffer at one time. The display can be viewed as a "window" that can be shifted right or left as needed to view the contents of the display buffer. When the display buffer contains information to the right of the viewing window, the triangular indicator in the upper *right* corner of the display is lighted. When the display buffer contains information to the left of the viewing window, the triangular indicator in the upper *left* side of the display is lighted. To see information to the right of the viewing window, hold down the right arrow key. To see information to the left of the viewing window, hold down the left arrow key.

The **display** statement also allows you to display more than one value at a time.

Enter	**Display**
markup=1.75	
display cost;markup	`10.25  1.75`
display cost;markup;cost+markup	`10.25  1.75  12`
display at(7),1;−2;3	`      1 -2  3`

When multiple values are **display**-ed, the values are known as a *print-list.* A print-list can contain both constants and variables, but must use a semicolon or a comma to separate the items of the list. The **display** statement operates differently depending upon the punctuation selected:

- When a *semicolon* print-list separator is used, the computer displays the items adjacently, as in the examples above. Remember, however, that BASIC automatically adds a space after every number and before each positive number.
- When a *comma* print-list separator is used, the computer displays the items beginning at certain column locations known as the display *zones.* These zones are 15 columns apart and correspond to display buffer positions 1, 16, 31, 46, 61, and 76. For example, try

Enter	**Display**
display cost,markup	`10.25          1.75`

The comma instructs the computer to display the next item in the print-list beginning at the next print zone (in this example, column 16). To view information displayed in the third, fourth, fifth, and sixth print zones, you must scroll the display window to the right using the [►] key.

Enter	**Display**
display 1,2,3,4,5,6	`1               2`

(Hold down the [►] key to view the information in print zones 3, 4, 5, and 6.)

The semicolon and comma separators can be mixed as desired in a print-list:

Enter	**Display**
display 1;cost,2;markup	`1  10.25          2  1.75`

Another option offered by the display statement is the **beep** parameter. The **beep** parameter can be used with or without a print-list and instructs the computer to sound an audible tone when executed. A comma must separate the **display** parameters from the information being displayed.

Enter	**Display**
display beep	
display at(12) beep,12	`           12`

The **beep** and **at** parameters must precede the print-list, if one is used, but can be entered in either order. The **at** parameter can be used only once in a **display** statement, but **beep** can be repeated as often as desired:

```
display beep beep beep beep beep
```

Other options available when using **display** are discussed in Lesson 20.

Technical Note: The **print** statement provides an alternate way of displaying information. Because the **print** statement has fewer options than the **display** statement, it is not discussed in this book.

For many examples of the **print** statement, refer to the *User's Guide* supplied with your computer.

LEARNED IN THIS LESSON

The **display** statement is used to place information in the display. The statement offers several options. You can display information starting at any column location up to 80 positions, sound a tone as information is displayed, and display more than one item of information at a time. If multiple items are displayed (a *print-list*), they must be separated by semicolons or commas. The semicolon instructs the computer to place items adjacently, and the comma instructs the computer to place items beginning at specific display locations called *zones*.

When the computer is in a pause state, the **CLR** or **ENTER** key must be pressed to allow the entry of additional instructions. (The computer shows a nonflashing underline cursor when you must press **CLR** or **ENTER** to proceed.)

Lesson 4

Beginning Programming

The preceding lessons illustrate using the computer in what is known as *immediate execution* or *command mode*. Such actions require the correct use of BASIC commands and statements, but they are not "programming." *Programming* consists of storing a set of instructions in the computer's memory. The stored instructions, or program, are not executed when you enter them, but only after you command the computer to **run** the program.

To store an instruction in memory, you must precede the instruction with a *line number.* A line number directs the computer to remember, but not execute, the BASIC instructions that follow.

Clear the computer's memory by typing **new** and pressing **ENTER**. Then type the following sequence exactly as shown:

```
    100 score=14
```

Before pressing **ENTER**, check that you have typed a space between the line number and the variable name. If not, press **CLR** and retype the sequence. When the line seems correct, press **ENTER**. The computer will store the "line" in memory and clear the display.

Technical Note: Compact Computer BASIC requires spaces after line numbers and between BASIC words. Spaces between other program elements are generally optional. The policy of this book is to show spaces only where absolutely required. If you feel uncor fortable with sequences such as score=14, enter score = 14. Ext spaces do not affect the computer's operation since it remove all unnecessary spaces when the program line is stored in memor Be certain, however, that you enter spaces wherever you see the in the text.

To confirm that the assignment statement has not yet been ex cuted, display the value of score:

Enter	**Display**
display score	0

Now execute the program by typing **run**, or pressing the RUN ke and pressing ENTER.

Enter	**Display**
run	

Although the computer's only apparent response is to clear tl display, the statement has been executed (assuming no error me sage was displayed). Check this by displaying the value of sco again.

Enter	**Display**
display score	14

This program, although very short, clearly illustrates the differen between command mode and programming. In command moc instructions are executed when ENTER is pressed. In programmir execution is deferred until the program is **run.**

The distinction between command mode and programming is al the basis for separating BASIC words into commands and stat ments. *Commands* are BASIC words that can be executed *o* in command mode. *Statements* are BASIC words that can be ex cuted in a program, and usually, in command mode as well.

The one-line program that you have just executed can be add to and made more useful by entering more program lines. For example, enter

```
110 display score
```

Notice that this line has a larger line number than the first. The computer uses the size (or magnitude) of the line numbers that you enter to determine the order in which program lines are executed. Since 110 is larger than 100, line 110 will be executed after line 100 when you run the program.

Run the program again.

Enter	**Display**
run	

Rather unexpectedly, the computer does not display the value of score. Why not? Actually the number 14 was displayed, but so quickly it was not visible. The computer automatically clears the display just before and immediately after running a program. Therefore, since the **display** statement is the last instruction in the program, its effect is automatically cleared when the program ends.

To instruct the computer to hold a value in the display until you have had an opportunity to see it, you must use the **pause** statement. The **pause** statement causes the computer to temporarily halt program execution without clearing the display. The computer remains in the pause state until you press ENTER or CLR to signal that program execution is to continue.

Add the following line to the program and run it again. The larger line number ensures that the **pause** statement will be executed after the **display** statement.

Enter	**Display**
120 pause	
run	14

This time the program works as expected, assigning and displaying the value of score. The computer then halts in the pause state with the underline cursor displayed, which indicates that you are expected to press ENTER or CLR. Program execution continues when you press one of these keys. In this case, of course, no additional program lines follow and the program ends (which clears the display).

Technical Note: Some BASICs require the use of an **end** statement at the end of a program. With Compact Computer BASIC, this statement is optional when the end of the program is the last line of the program. It is possible, however, to design a program to end at a line number other than the last. When this occurs, an **end** statement is no longer optional. In this book, **end** statements are omitted except where required as explained above. See Lesson 19 for an example of a program that requires an **end** statement.

The preceding program illustrates one of the main advantages of programming your instructions over executing them in the command mode: they can be easily executed more than once. To reexecute these instructions, **run** the program again. The program will remain in memory until you clear it by executing a **new** command. Even turning the computer OFF and ON will not erase it. (Of course, the contents of memory will be lost if power is removed as a result of discharging or removing the batteries.)

As you work through this book, you will write programs that are longer and more complicated. When a program grows beyond five or six lines, it becomes difficult to remember what instructions you have already entered. Fortunately, BASIC has a command to allow you to view or **list** program instructions.

The **list** command has two forms: **list** and **list** *line-group.* When **list** is executed, the computer displays the lowest numbered program line and halts. You can then display the next program line by pressing ENTER or ▼, or clear the display and end the listing operation by pressing CLR.

The sole difference in executing **list** with a line-group parameter is that the listing begins with the line number specified by the line-group. A line-group can be either a single line number or a range of line numbers specified by a beginning line number, a hyphen, and an ending line number. If only a beginning line number and a hyphen are entered, the ending line number is assumed to be the last line in memory. (The hyphen is entered with the minus key.)

You still press ENTER or ▼, to advance to the next line number and CLR to end the listing.

To illustrate the **list** command, **list** the score program.

Enter	**Display**
list	100 SCORE=14
ENTER	110 DISPLAY SCORE
ENTER	120 PAUSE
ENTER	

Notice that the computer automatically translates BASIC words and variable names from lowercase letters into capital letters.

Before ending this lesson, it is important to cover several additional points regarding BASIC line numbers. First, observe that we began this program with line number 100 and incremented each line by 10. The actual numbers selected were unimportant. A functionally equivalent program could have been written using any three line numbers in the valid range of 1 through 32766, *as long as the lines remained in the same relative order.*

Line numbers should not normally follow each other in increments of 1, however. By selecting line numbers with unused values or "gaps" between them, you make it easy to insert additional program lines. Simply enter the new line with a line number between the existing line numbers.

For example, suppose you want to increase the value of score by 7. This is accomplished by entering the following line:

```
105 score=score+7
```

Confirm that the line has been inserted between lines 100 and 110 by listing the program again.

Enter	**Display**
list	100 SCORE=14
ENTER	105 SCORE=SCORE+7
ENTER	110 DISPLAY SCORE
ENTER	120 PAUSE

Another important point regarding line numbers is the ease with which they allow program instructions to be replaced or deleted. To replace a program line, simply enter a new line with the same line number. Since BASIC does not allow a program to have two lines with identical line numbers, the computer automatically replaces the original line with the new line. Test this feature by replacing line 105 with

```
105 score=score+14
```

List the program again.

Enter	**Display**
list	100 SCORE=14
ENTER	105 SCORE=SCORE+14
ENTER	110 DISPLAY SCORE
ENTER	120 PAUSE

The original line has been replaced with the new one.

Program lines are deleted by the **delete** *line-group* command. (As with the **list** command, a line-group can be either a single number or a range of numbers separated by a hyphen.) This command, which can be shortened to **del** to reduce typing, instructs the computer to erase a line or a range of lines. For example, **del** 100 will erase line 100; **del** 100-120 will erase lines 100 through 120; **del** -200 will erase all lines from the beginning of memory to line 200; and **del** 200- will erase all lines from 200 to the end of memory. You can test this command by entering:

```
del 100-
```

If you now enter **list**, no lines will be displayed.

LEARNED IN THIS LESSON

Every BASIC program, from the simplest to the most complicated, consists of one or more numbered "lines" of BASIC instructions. The numbers that precede each line serve three purposes:

1. They allow the computer to distinguish between instructions to be stored in memory (a program) and instructions to be executed immediately (command mode).
2. They determine the order in which program instructions are executed.
3. They provide a method of identifying program instructions that you can use when listing or editing a program.

Line numbers must be whole numbers in the range 1 through 32766. As a rule, select line numbers that are 10 or more units apart. This practice makes it easy to insert additional lines.

The **list** command is used to display program lines. The command can be used singularly or with a line-group parameter. When **list** is used by itself, the listing begins with the lowest numbered program line. When used with a line-group parameter, the listing begins and ends with a specified line number. To advance to the next program line, press [ENTER] or [▼]. To clear the display and end the listing, press [CLR].

The **delete** line-group command is used to delete specific program lines. The command can be abbreviated **del**, if desired.

The **pause** statement should be placed in a program whenever you want displayed information to remain in the display. The **pause** statement instructs the computer to halt program execution without altering the display until [ENTER] or [CLR] is pressed.

Lesson 5

Strings—Alphanumeric Information

The most significant feature of a computer is not its capability to manipulate numbers, since a pocket calculator can do that, but its additional capability to manipulate letters, symbols, signs, words, sentences, and, in general, textual information of any type. To enable the computer to distinguish between textual information and numbers, *textual information must be entered into the computer enclosed in quotation marks.* Such information, which can consist of any collection of alphabetic letters, signs, symbols, or digits from 0 to 255 characters in length, is referred to as a *string.* For example, enter

Enter	**Display**
display "CC-40"	CC-40

(Use the SHIFT key to enter the capitals and the minus − key to enter the hyphen.)

Observe that the computer displays exactly what you enter between the quotes, but not the quotes. Notice also the familiar underline cursor in column one that indicates the computer is in the pause state.

As you might suspect, a string such as that illustrated above is referred to as a *string constant.* Like a numeric constant, it can be changed only by retyping. You can create *string variables,* however, by simply assigning a string to a variable name that has as its *last* character a dollar ($) sign.

Enter	**Display**
c$="Computer"	
display c$	Computer

If you attempt to assign a string to a variable name that does not end in a dollar sign, you will get an error message. Other than this requirement, the rules governing the creation of string variable names are the same as those for numeric variable names: they must begin with a letter, underline, or @ symbol; they cannot be more than 15 characters in length (including the $ sign); and they cannot be on the reserved word list.

As you would expect, string constants and variables can also be displayed in a print-list, and can be intermixed with numeric constants and variables.

Enter	**Display**
name$="CC-40"	
display name$;c$	CC-40Computer
display 1;name$;c$	1 CC-40Computer

Notice that the computer does not automatically add spaces before and after strings as it does for numbers. If you want spaces displayed between the strings, you can include the spaces in the appropriate places within the strings:

Enter	**Display**
name$="CC-40 "	
display 1;name$;c$	1 CC-40 Computer

Or set a string equal to a single space and display it or a space constant between print items.

Enter	**Display**
name$="CC-40"	
space$=" "	
one$="1"	
display one$;space$;name$;" ";c$	1 CC-40 Computer

As stated above, a string can be from 0 to 255 characters in length. That a string has a maximum length is logical, but you may well wonder how a string can consist of zero characters. Such a string is referred to as a *null* string and is created by the sequence "" (no space between the quotes). (Lesson 24 illustrates several applications of the null string.)

A string, even one consisting entirely of digits, can never be used as a number. If you attempt to perform a mathematical operation using a string or string variable, you will get an error message.

As you become familiar with BASIC, you will encounter the use of the dollar sign regularly. The dollar sign always refers to a string or string operation and is pronounced "string." For example, the variable *name$* is pronounced "name string," not "name dollar."

One of the more interesting operations that can be performed with strings is called *concatenation*. Concatenation is simply a large word for the process of joining strings together. The ampersand (&) symbol is used to instruct the computer to concatenate strings.

Enter	**Display**
a$=one$&" "&name$&" "&c$	
display a$	`1 CC-40 Computer`

Concatenation always adds strings in left to right order.

Enter	**Display**
z$=c$&space$&name$	
display z$	`Computer CC-40`

A sequence of strings joined by the concatenation symbol is referred to as a *string-expression*.

LEARNED IN THIS LESSON

Textual information can be manipulated by the computer if it is entered in the form of *strings*. Strings must be enclosed in quotation marks and can consist of any collection of alphabetic letters, signs, symbols, or digits from 0 to 255 characters in length.

Strings can be either *string constants* or *string variables.* String variables must conform to the same rules as numeric variables with the added restriction that the last character of the variable name must be a dollar sign.

A string of zero length is a special string referred to as a *null* string.

Strings can be joined together using the ampersand (&) symbol. This operation, referred to as *concatenation,* combines strings in left to right order. A sequence of strings joined by the concatenation symbol is referred to as a *string-expression.*

Review Test 1

1. Which of the following are valid variable names?

   (a) dollar$
   (b) acreinch
   (c) AVE.
   (d) start
   (e) z1z1z
   (f) FIRST#
   (g) @$
   (h) ØO
   (i) PHd$
   (j) CC-40
   (k) end
   (l) phantasmagoria$
   (m) enter
   (n) 1st_count

2. What are the rules for creating variable names?

3. Which of the following are valid assignment statements?

   (a) A$=""
   (b) 2 t$=","
   (c) 100meters=feet*0.3048
   (d) 8=4+4
   (e) let g$="exo"&"gust"
   (f) 12 F=F−F
   (g) pay=wages*rate
   (h) 2=t
   (i) move$="P-K4"
   (j) 50 word$ = old$&new$

4. What is the initial value of a numeric variable? What is the initial value of a string variable?

5. How large is the display buffer? How much of it can you view at one time?

6. What does the statement **display** 1,,3 do? Is this a legal sequence?

7. What is the function of the comma separator in a print-list? What is the function of the semicolon in a print-list?

8. Which of the following do you think are valid **display** statements?

   (a) display at(0),x
   (b) display "tes";;;;;"t"
   (c) display beep at(12) beep,"mass"
   (d) display ,,,,,,,7
   (e) display at(7),7 at(11),11
   (f) display "1,3"
   (g) display display

9. Can 0 be a line-number?

10. How can you insert a new program line in a program? Is it possible to write a program in which a new line cannot be inserted?

11. How do you replace a line?

12. How do you erase a program? How do you erase a line?

13. Do you think the following program will function as the writer intended?

```
10 a$="Humphrey "
20 b$="Bogart"
30 display a$&b$
```

Lesson 6

INPUTing Keyboard Information

The word "input" refers to the entry of information into a computer *during the execution of a program.* Naturally enough, the BASIC statement for this operation is **input**.

Clear the computer's memory by entering **new** and then type in the following illustrative program.

```
100 input a
110 display "a=";a
120 pause
```

Run the program.

Enter	**Display**
run	?

The question mark that you see is generated by the **input** statement and is an example of a *prompt.* A prompt is a sign or message displayed by the computer (or program) indicating that it is waiting for information. Respond to this prompt by typing 3313 and pressing [ENTER].

Enter	**Display**
3313	a= 3313

The program accepts what you type and assigns it to the variable that follows the **input** statement. (The variable after **input** is *not* optional.) Next the program displays the value of the variable ar pauses. The **input** statement, then, is an assignment stateme that allows you to enter the value of a variable during progra execution.

As an experiment, run the program again and enter the word "tes in response to the prompt.

Enter	**Display**
run	?
test	Variable not defined

The computer displays an error message because you did not ent the kind of information that can be assigned to a numeric variabl The error has not stopped program execution, however. Pres ENTER and the ? prompt will be displayed again. In fact, until yo enter a number or press the BREAK key, the computer will refus to proceed beyond the **input** statement.

If you want to input string information, use a string variable followin the **input** statement:

```
100 input a$
110 display "a$=";a$
```

(These lines replace previous lines 100 and 110.)

You can now enter string information without creating an error cond tion.

Enter	**Display**
run	?
qwerty	a$=qwerty

A prompt such as a question mark is too general for most applica tions. While it does indicate that you are expected to enter some thing, it gives no hint as to what information to enter. As a rule, is better to provide prompts that describe what the program use is expected to input.

To facilitate the display of descriptive prompts, the **input** statemer has a built-in display capability. You can use this feature by placin the prompt that you want to display in quotation marks and locating it *after* the **input** statement and *before* the input variable. *A semicolon must separate the prompt from the variable, and the prompt must be less than 31 characters long or only the first 30 characters are displayed.*

Example:

```
new      (To clear memory)
100 input "Enter a number ";a
110 display "The number is";a
120 pause
130 input "Enter a string ";a$
140 display "The string is ";a$
150 pause
```

After listing the program to be certain that you have entered it correctly, **run** it.

Enter	**Display**
run	`Enter a number`
186284	`The number is 186284`
[ENTER] *(To end pause)*	`Enter a string`
first violin	`The string is first violin`
[ENTER] *(To end pause)*	

Observe that the computer does not display a question mark when you use a prompt with the **input** statement. Since you have created your own prompt, the computer assumes that you have included a question mark if you want one. In this case, the prompt is better without one.

Run the program several times. The **input** statement will be used in nearly every program you write so you will need to be certain you understand its operation.

Enter	**Display**
run	`Enter a number`
−35.01	`The number is -35.01`

(*cont. on following page*)

(cont. from preceding page)

Enter	Display
ENTER	Enter a string
Joe Green	The string is Joe Green
ENTER	
run	Enter a number
11+33	The number is 44
ENTER	Enter a string
,	Bad INPUT data
ENTER	Enter a string
A PENNY	The string is A PENNY

Notice that the **input** statement allows you to enter mathematical expressions (calculations) as well as constants for numeric information, but it does not allow you to enter a comma for string information.

The **input** statement also allows you to correct a mistake by pressing CLR and retyping if you have not pressed ENTER yet:

Enter	Display
run	Enter a number
911 CLR 119	The number is 119
ENTER	Enter a string
texas CLR Texas	The string is Texas

The **input** statement has another useful feature: A single **input** statement can be used to display multiple prompts and accept multiple inputs. To use this feature, you must separate the additional prompts with a comma. The following program uses this feature of the **input** statement to compute the volume of a box.

```
new
100 input "Length? ";length,"Width? ";width,"Height? ";height
110 volume=length*width*height
120 display "Volume =";volume;"cubic feet"
130 pause
```

Run the program. (The values entered are assumed to be in feet.)

```
Enter    Display
run      Length?
97       Width?
17       Height?
9.5      Volume = 15665.5 cubic feet
```

The computer automatically displays the next prompt in the sequence when you enter the expected input.

The **input** statement also allows the use of string variables or expressions as prompts. The volume program is revised below to illustrate this feature.

```
new
100 e$="Enter "
110 l$="length: "
120 w$="width: "
130 h$="height: "
140 input e$&l$;length,e$&w$;width,e$&h$;height
150 volume=length*width*height
160 display "Volume =";volume;"cubic feet"
170 pause
```

A sample **run** is shown below.

```
Enter    Display
run      Enter length:
7        Enter width:
11       Enter height:
3        Volume = 231 cubic feet
```

Although a program is usually better if descriptive prompts are used, the prompt parameter is always optional. If desirable, the **input** statement can accept multiple inputs without displaying prompts. The following program illustrates this option.

```
new
100 input n1,n2,n3,n4,n5
110 display n1;n2;n3;n4;n5
120 pause
```

A comma must separate the additional input variables.

Enter	**Display**
run	?
1	?
2	?
3	?
4	?
5	1 2 3 4 5

Technical Note: The **input** statement is also used to read informa tion from a peripheral device. This application of the **input** statemen is beyond the scope of this book. Refer to the *User's Guide* supplie with your computer for information on this subject.

LEARNED IN THIS LESSON

The purpose of the **input** statement is to allow the entry of numeric or string information during program execution. The information entered is assigned to the variable following the **input** statement. The type of input variable used (numeric or string) determines the type of information accepted by the **input** statement.

If used without any of the optional parameters, the **input** statement displays a question mark *prompt* when executed. A prompt is a sign or message indicating that the computer is waiting for keyboard information. To permit a program to display more informative prompts, the **input** statement has a built-in display capability. The general form of the **input** statement when used with the prompt option is

input *prompt*;*variable*

A semicolon must separate the prompt from the variable, and the prompt must be less than 31 characters long or only the first 30 characters are displayed. The prompt can be either a string constant or a string variable.

The **input** statement can display multiple prompts and accept multiple inputs. The general form of the **input** statement when used with multiple prompts is

Lesson 7

More on the PAUSE Statement

The purpose of the **pause** statement is to hold information in the display until it can be viewed. The statement can be used by itself, as illustrated in Lessons 4 and 6, or with one of two optional parameters.

The optional forms of the statement are

> **pause** *numeric-expression*
>
> **pause all**

The *numeric-expression* parameter allows you to specify the number of seconds that information is paused in the display to 0.1 second accuracy. This form of the **pause** statement, called a *timed pause,* eliminates the requirement to press [ENTER] or [CLR] to continue program execution. After the specified time value has elapsed, the computer automatically executes the next program statement.

As indicated by the general term "numeric-expression," the time value in a timed **pause** statement can be a numeric constant, variable, or calculation. This value can also be either positive or negative. If a positive value is specified, you can override the timed **pause** by pressing [ENTER] or [CLR]. If the value is negative, the timed **pause** cannot be overridden.

The following program provides an example of the operation of the timed **pause** statement. Written for a retail business that calculates its selling prices by adding a markup percentage to the wholesale costs of merchandise, the program calculates and displays the dollar amount of the markup and the final selling price.

```
new
100 input "Wholesale cost? ";cost
110 input "Markup percentage? ";percentage
120 markup=cost*percentage/100
130 display "Markup =";markup
140 pause 3.5
150 display "Selling price =";cost+markup
160 pause 3.5
```

An illustrative **run** of the program is shown below.

```
Enter     Display
run       Wholesale cost?
21.14     Markup percentage?
36        Markup = 7.6104
          Selling price = 28.7504
```

The **pause all** version of the **pause** statement instructs the computer to execute an automatic, untimed **pause** after every **display** statement. A **pause all** statement takes effect as soon as it is executed, and remains in effect until cancelled by the execution of a **pause** 0 statement.

The major advantage of the **all** parameter is that it reduces program size by removing the need to place **pause** statements later in the program. The retail markup program is rewritten below to illustrate the operation of the **pause all** statement.

```
new
100 pause all
110 input "Wholesale cost? ";cost
120 input "Markup percentage? ";percentage
130 markup=cost*percentage/100
140 display "Markup =";markup
150 display "Selling price =";cost+markup
```

input *prompt*;*variable*,*prompt*;*variable*,*prompt*;*variable* . . .

A comma must separate the additional prompts.

If desirable, the **input** statement can accept multiple inputs without displaying a prompt by using the form:

input *variable*,*variable*,*variable* . . .

Since the prompt option is not being utilized, the **input** statement displays a question mark for each input variable.

A sample **run** of the new version of the program is shown below.

Enter	**Display**
run	`Wholesale cost?`
21.14	`Markup percentage?`
36	`Markup = 7.6104`
ENTER	`Selling price = 28.7504`
ENTER	

LEARNED IN THIS LESSON

The two optional forms of the **pause** statement are

pause *numeric-expression*

pause all

The *numeric-expression* option allows a program to pause information in the display for a variable time period, depending upon the value of numeric-expression. This value can be entered as a numeric constant, variable, or calculation. Program execution continues automatically after the specified time period has elapsed.

The **pause all** statement instructs the computer to execute an automatic, untimed **pause** after every **display** statement. This option requires that you press ENTER or CLR to continue program execution after each **display** statement, just as if a regular **pause** followed the statement. The **all** option remains in effect until a **pause** 0 statement is executed.

Lesson 8

Program Branching—The GOTO Statement

The normal sequence of program execution is from the smallest to the largest line number. This order is easily altered, however, by the use of the **goto** statement. The **goto** statement has the general form:

> **goto** *line number*

The **goto** statement instructs the computer to transfer program execution from the current program line to the line number, or *transfer address,* following the **goto** statement. For example, executing

```
50 goto 10000
```

transfers program execution from line 50 to line 10000, skipping all program lines in between. Of course, the transfer address must exist or an error condition will occur.

The advantage of the **goto** statement is that it allows a program to control the order in which program lines are executed. If necessary, a program can skip the execution of a line or group of lines, or, as in the next example, repeat a sequence of lines.

Enter and **run** the following program.

```
new
100 x=0
110 display x
```

(*cont. on following page*)

```
(cont. from preceding page)
120 x=x+1
130 goto 110
```

The computer begins madly counting by 1s. The program tells the computer to display the value of x, add 1 to x, and then go to line 110, which repeats the sequence. A program segment that is executed over and over such as this is known as a *loop.*

In this example, the computer has been placed in an *infinite loop.* It will execute lines 110, 120, and 130 until either its batteries discharge or . . . ?

Fortunately computer manufacturers are aware of the ease with which computers can be placed in infinite loops and have provided a quick method of manually halting or "breaking" a program—the BREAK key. The BREAK key instructs the computer to stop what it is doing and return to command mode.

To satisfy yourself that you can always stop a running program, **run** and break this program several times. You will notice that whenever you press BREAK the computer displays `Break` followed by the familiar underline cursor. Of course you must press CLR or ENTER before you can perform another operation.

Observe that no **pause** statement is used in this program. The period of time for which the value of x is displayed is determined by the time it takes the computer to increase x by 1 and return to line 110. If you want to slow the counting down, insert a **pause** statement between line 110 and line 120:

```
115 pause .3
```

When you now **run** the program, it counts more slowly.

Here is another interesting modification: Break the program and replace line 110 with

```
110 display x;
```

(The program as it should now look is listed below. **List** your program to ensure that you have made no entry mistakes.)

Enter	Display
list	100 X=0
ENTER	110 DISPLAY X;
ENTER	115 PAUSE .3
ENTER	120 X=X+1
ENTER	130 GOTO 110

Run the program. Now, instead of counting in one place in the left side of the display, the computer displays successive values horizontally. After displaying 9, the computer pauses for several seconds before continuing the count in the left side of the display with 22. If you continue to watch, the pause repeats after 22, 42, 62, 82, ... Although you cannot see the numbers to the right of the display, what is occurring is shown in Fig. 8-1. The delay is explained by the time it takes the computer to place values 10 through 21 in the portion of the display buffer to the right of the current viewing window. Once the 80-character display buffer is filled, the computer clears the buffer and starts again at the left side of the display. The part of the display buffer that you actually see is the left-most 31 characters.

You can check the validity of this explanation as follows: Break the program during the delay period. Then press PB (SHIFT ▲) twice. The display should show the values of x. By now scrolling the viewing window with the ► key, you can view the contents of the display buffer to the right of column 31.

As another experiment, replace line 110 with

```
110 display x,
```

When you now **run** the program, the computer displays the values of x in print zones, rather than adjacently as specified by the semicolon.

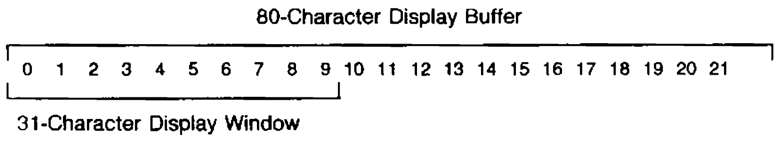


Fig. 8-1 Illustration showing the effect of putting a semicolon after the **display** statement.

Whenever you place a semicolon or a comma after a **display** statement, you create a *pending display state.* A pending display state causes the computer to preserve the previous display contents when the next **display** statement is executed, unless the display buffer is full. When the buffer is full, it is cleared and displaying begins anew at column 1.

Counting examples serve primarily to demonstrate visually the looping power of the **goto** statement. A more practical application of a loop is shown below.

```
new
100 pause all
110 input "Feet? ";feet
120 meters=feet*.3048
130 display "Meters =";meters
140 goto 110
```

This program performs a simple function. It takes the value you enter in response to "Feet? " and converts it to meters. By performing the function in a loop, however, you avoid having to reenter **run** to convert additional values. The program automatically loops back to the input prompt until you press the BREAK key to stop program execution.

Enter	Display
run	`Feet?`
44	`Meters = 13.4112`
ENTER	`Feet?`
99.9	`Meters = 30.44952`
BREAK	`Break`

This technique is often used when an application requires repetitive work.

LEARNED IN THIS LESSON

The **goto** statement instructs the computer to transfer program execution from the current line to the line number following the **goto** statement.

The **goto** statement can be used to skip the execution of a line or group of lines, or to place a program in a *loop.* Most programs contain at least one loop, although as a rule not an *infinite loop* as in the examples in this lesson. An infinite loop is a sequence of program instructions that are executed over and over without any provision for exit. The decision-making capabilities of the computer discussed in Lesson 10 make it possible to write programs that control the number of times that a loop is executed.

An infinite loop can always be stopped manually by pressing $\boxed{\textbf{BREAK}}$. The $\boxed{\textbf{BREAK}}$ key will stop any running program, not just a program in an infinite loop.

Placing a semicolon or a comma at the end of a **display** statement instructs the computer to preserve any information presently in the display when the next **display** statement is executed. This is referred to as a *pending display state.* Once the entire 80-character display buffer is full, it is cleared and displaying begins anew at column 1.

Lesson 9

The NUMBER and RENUMBER Commands

Although it is necessary to begin each program line with a line number, it is not necessary to type these numbers yourself. BASIC provides a command to automatically number program lines as you enter them—the **number** command. This command has the general form

number *initial-line,increment*

where *initial-line* is the line number you want assigned to the first program line and *increment* is the amount by which you wish the lines to increase. If you enter no values for initial-line or increment, the computer assumes an initial-line of 100 and an increment of 10. The command can be abbreviated **num**, if desired.

To illustrate this command, clear the computer's memory and enter **num**.

Enter	Display
new	
num	100

The computer places the number 100 in the display and then waits for you to enter a program statement:

Enter	Display
100 x=1	110

(You enter only the x=1, the computer provides the 100.)

When you do enter a statement, the computer displays the next line number and again waits for your entry:

Enter	Display
110 display x	120

This process repeats for as long as you enter program statements. When you are done entering your program, simply press ENTER or BREAK in response to the next line number and the automatic line numbering will end:

Enter	Display
120 x=x+1	130
130 goto 110	140
ENTER	

A listing of the program as you should have entered it is shown below.

```
100 X=1
110 DISPLAY X
120 X=X+1
130 GOTO 110
```

By omitting both the initial-line and the increment in this example, you instructed the computer to use the default values of 100 and 10, respectively. If you want your program to begin with another initial value or increment at a different rate, then substitute the values you want. The following table shows several **number** examples.

Command	Meaning
num 1	Number by tens beginning with line 1
num ,50	Number by 50s beginning with line 100
num 11,121	Number by 121s beginning with line 11

If you are using the **number** command and the computer comes to a line number that already exists, it displays that line and you can either press ENTER to advance to the next line number, change the displayed line, or press **BREAK** to cancel the numbering operation. (The **number** command will not display a line that already exists if the number of the line is not evenly divisible by the specified increment.)

The **number** command is convenient for numbering program lines while entering a program. To change line numbers after a program has already been entered, use the **renumber** command:

> **renumber** *initial-line,increment*

Like the **number** command, **renumber** lets you specify an initial-line and an increment. If you enter no values for these parameters, the computer assumes an initial-line of 100 and an increment of 10. The **renumber** command can be abbreviated **ren**.

To illustrate the **renumber** command, change the line numbers of the counting program that you just entered so that it begins with line 2000 and increments by 3 (if you have cleared the program, you will have to reenter it before you do this):

```
ren 2000,3
```

Now **list** the program.

Enter	**Display**
list	`2000 X=1`
ENTER	`2003 DISPLAY X`
ENTER	`2006 X=X+1`
ENTER	`2009 GOTO 2003`
ENTER	

Not only does **renumber** change the numbers of program lines, it also changes all references to program lines within the program. Thus, the **goto** 110 of the original program is changed to **goto** 2003 in the renumbered version. Without the capability to change references within a program, the **renumber** command would be much less useful.

If you **renumber** a program that has a reference to a line number that does not exist, the computer renumbers the program as requested, but it also displays the message `Line not found` and gives the unexplained line reference the value 32767. The computer always substitutes the value 32767 so that you can easily find the erroneous line. To test this, change line 2009 in this example to

```
2009 goto 1
```

and **renumber** the program again.

Enter	**Display**
ren	`Line not found`
ENTER *(clears message)*	
list	`100 X=1`
ENTER	`110 DISPLAY X`
ENTER	`120 X=X+1`
ENTER	`130 GOTO 32767`

The computer could find no program line 1, so it replaced **goto** 1 with **goto** 32767. The program will not run correctly until you change this statement to **goto** 100.

LEARNED IN THIS LESSON

The **number** command instructs the computer to automatically number program lines as you enter them. The general form of the command is

number *initial-line,increment*

If no initial-line or increment is given, the computer assumes values of 100 and 10, respectively. To cancel automatic line numbering, press ENTER in response to a blank line number or press BREAK at any time.

The **renumber** command changes line numbers after a program has been entered. The command changes both line numbers and references to line numbers within the program. The **renumber** command also allows you to specify an initial-line and increment in the form

renumber *initial-line,increment*

If no initial-line or increment is given, the computer assumes values of 100 and 10, respectively.

The **number** command can be abbreviated **num** and the **renumber** command can be abbreviated **ren**.

Lesson 10

Decision Making—The IF THEN Statement

The program statements discussed so far have all had one characteristic in common: they are always executed when encountered in a program. With the **if then** statement, however, the execution of a statement, line, or group of lines can be based upon the result of a decision-making test.

The general form of the **if then** statement is

if *condition* **then** *action*

where *condition* consists of a decision-making test and *action* is any valid program statement.

The decision-making tests discussed in this lesson are *relational* tests. Lesson 26 discusses the use of *logical* decision-making tests.

Six relational tests are available. The symbols used to perform them in BASIC and their meanings are listed below. Notice that the not-equal-to, less-than-or-equal-to, and greater-than-or-equal-to symbols are made by combining two other symbols. (The more usual symbols for these operations, ≠, ≤, and ≥, respectively, are not present on most computer keyboards.)

Symbol	Meaning
<	less than
>	greater than

(*cont. on following page*)

(*cont. from preceding page*)

Symbol	**Meaning**
=	equal to
<>	not equal to
<=	less than or equal to
>=	greater than or equal to

A relational test can have only two possible results: either the relation is *true*, or it is *false*. If the relation is true, then the action is performed. If the relation is false, the action is skipped. This rule can be summarized as "do if true, skip if false."

The following program illustrates how the **if then** statement is combined with a relational test to make program decisions. After setting up the automatic pause, the program requests that you enter a number. When you do, the **if then** statement compares the number with zero. If your entry is less than zero (relation true), the **display** statement is executed. If your entry is not less than zero (relation false), the **display** statement is skipped.

```
new
100 pause all
110 input "Enter a number: ";n
120 if n<0 then display "The number is negative!"
130 goto 110
```

Run the program and try various numeric entries. Whenever a value less than zero is entered, the program displays "The number is negative!"

Enter	**Display**
run	`Enter a number:`
−7	`The number is negative!`
ENTER	`Enter a number:`
7	`Enter a number:`
BREAK	`Break`

As it stands, the program only informs you when a number is negative. With several additional decision-making tests, you can make the program tell you whether your entry is negative, positive, or zero.

```
122 if n>0 then display "The number is positive!"
124 if n=0 then display "The number is zero!"
```

Run the expanded version of the program.

Enter	**Display**
run	`Enter a number:`
99	`The number is positive!`
ENTER	`Enter a number:`
−.001	`The number is negative!`
ENTER	`Enter a number:`
0	`The number is zero!`
BREAK	`Break`

This program analyzes the number that you enter by "filtering" it through three decision-making tests. Since the three tests are mutually exclusive, only the **display** statement following the true test is executed. The other **display** statements are skipped.

Technical Note: It is the action that follows the **then** part of the **if then** statement that is skipped if the relation is false, not the relational test. The relational test is always performed.

The next program illustrates how an **if then** statement can be used to control the number of times a loop is executed. The program calculates the average of a series of numbers that you input.

```
new
100 total=0
110 input "Number of entries? ";n
120 count=n
130 input "Entry? ";entry
140 total=total+entry
150 count=count-1
160 if count=0 then goto 180
170 goto 130
180 average=total/n
190 display "Average =";average
200 pause
```

The program begins by asking how many entries there are and assigning that value to the variable count. Next the program requests your entry. The value that you enter is added to the variable total. Then, after subtracting 1 from count, the program tests if count equals zero. If count does equal zero (relation true), execution is transferred to line 180 to calculate and display the average. If count does not equal zero (relation false), the conditional action is skipped, allowing execution to advance to line 170—which sends the program back to line 130.

A sample **run** of the program is shown below.

Enter	**Display**
run	`Number of entries?`
5	`Entry?`
10	`Entry?`
20	`Entry?`
30	`Entry?`
40	`Entry?`
50	`Average = 30`

This program illustrates a common programming technique. The computer is placed in a loop until a counting variable reaches a desired terminating value, and then program execution is transferred out of the loop. The purpose of the counting variable is to control the number of times the loop is executed. In this example, the counting variable is set initially to the number of entries and is decreased by 1 each time a new entry is accepted (a "countdown").

An interesting point to observe is that the decision used to end the loop can be written two ways. In the current example, the program tests whether the value of count equals zero. By modifying the program to test whether the value of count is *not* equal to zero, the program can be made shorter and more efficient.

```
new
100 total=0
110 input "Number of entries? ";n
120 count=n
```

```
130 input "Entry? ";entry
140 total=total+entry
150 count=count-1
160 if count<>0 then goto 130
170 average=total/n
180 display "Average =";average
190 pause
```

This program is functionally equivalent to the previous program. Instead of using the **if then** statement to transfer program execution out of the loop, this version of the program uses the **if then** statement to keep the program looping until the exit condition is met.

In both of these examples, the action following the **if then** statement was a **goto** statement. One of the features of BASIC is that when a **goto** statement is used as a conditional action, the **goto** can be dropped and only the line number need be supplied. Thus, a statement such as

```
160 if count<>0 then goto 130
```

can also be entered as

```
160 if count<>0 then 130
```

BASIC recognizes the implied **goto** between **then** and 130.

LEARNED IN THIS LESSON

The **if then** statement provides a BASIC program with the capability to perform different actions depending upon the result of a decision-making test. If the test is true, the action following the **then** portion of the **if then** statement is executed. If the test is false, the action is skipped.

The decision-making tests discussed in this lesson are listed below.

Symbol	**Meaning**
<	less than
>	greater than

(*cont. on following page*)

(*cont. from preceding page*)

Symbol	Meaning
=	equal to
<>	not equal to
<=	less than or equal to
>=	greater than or equal to

A common application of decision making is to control the number of times that a loop is executed. The usual technique is to maintain a counting variable that is either increased or decreased by 1 each time the loop is executed until a desired terminating value is reached. When the counting value reaches the terminating value, execution is transferred out of the loop with a **goto** statement.

When a **goto** statement is used as the action parameter of an **if then** statement, the **goto** can be dropped and only the line number entered. For example, both

```
100 if a=10 then 150
```

and

```
300 if a<>120 then 90
```

are valid **if then** statements that use this option. The computer understands the implied **goto** in these lines.

Review Test 2

1. Which of the following are valid **input** statements?

   (a) 5 input ENTER YOUR NAME:;n$
   (b) 300 input a;b;c;d;e
   (c) 110 input "Account? ";account+2
   (d) 1 input ID$&"−001";item$,"quantity";quantity
   (e) 50 input "Enter five names:";N1$,N2$,N3$,N4$,N5$
   (f) 22 input "Direction? ",d$

2. How do you cancel a **pause all** statement in a program?

3. Write a program that counts by 9s and displays the count in column 9 of the display. Modify the program to count backwards by 99 beginning at 2475. Make the second program **beep** every time it displays a number.

4. What is a *pending display state?* How is it created and why is it useful?

5. The first two lines of a program designed to balance a checking account are given below. Complete this program so that you can use it to balance your checkbook. Assume that deposits are entered as positive values and checks as negative values.

```
100 INPUT "Current balance? ";BALANCE
110 INPUT "Amount of check or deposit? ";AMOUNT
```

6. What is a "counting variable" and how is it used to control the number of times a loop is executed?

7. Which of the following are valid **if then** statements?

   (a) 100 if x=x+1 then 300
   (b) 121 if true=false then false=true+2
   (c) 112 if T><12 then input "ENTER A NEW NUMBER";T
   (d) 50 if x<y<z then 200
   (e) 75 if 2*B−C/12<=2400/C then ID$="end"
   (f) 92 if count + 1 = 99 then goto 2001
   (g) 100 if yards>=35 then if down=4 then display "BETTER KICK!"

8. A wholesale distributor of chess sets offers the following discount schedule: 20% off on orders of 100 or fewer sets, 25% off on orders between 101 and 500 sets, 30% off on orders between 501 and 1000 sets, 35% off on orders between 1001 and 4999 sets, and 40% off on orders of 5000 or more sets. The base cost of the chess sets is $14.88 per set. Write a program that allows you to input the number of chess sets ordered and displays the total cost of the order. Use the program to calculate the wholesale cost of 433 sets, 888 sets, and 2001 sets.

Lesson 11

Order of Calculations

A fundamental rule of mathematics is that a series of mathematical operations can have only one result. Without such a convention, problems which intermix different mathematical operations could have several answers, depending upon the order in which the operations are performed. Consider, for example, the problem

$$3 + 5 \times 2 + 4 =$$

If the addition operations are performed before the multiplication operation, the result is $8 \times 6 = 48$. If the multiplication operation is performed before the addition operations, the result is $3 + 10 + 4 = 17$.

The correct answer to this problem according to the rules of mathematics is 17, the answer given by the computer if you enter the calculation just as it is written. To obtain this answer, the computer must perform multiplication before addition. This capability to perform one operation before another is achieved by a built-in ranking system and a set of rules governing the order in which operations are executed. These rules are usually referred to as *levels of precedence.*

The levels of precedence used by the computer in order of highest to lowest priority are

Precedence	Operation Name	Operator Symbol
1	Higher-order functions	(see discussion below)
2	Exponentiation	∧
3	Unary minus (negation)	−
4	Multiplication and division	* and /
5	Addition and subtraction	+ and −

The higher-order functions assigned to the first precedence level consist of mathematical operations such as common and natural logarithms, trigonometric functions, and absolute value, as well as operations unique to BASIC such as **int**, **rnd**, and **len**. Because of the variety and complexity of these functions, they are discussed in later lessons (see Lessons 12, 13, 14, and 25). For the moment, be aware that when intermixed with other operations from this table, higher-order functions are performed first.

The second priority level is assigned exponentiation. Exponentiation means "raised to a power" and refers to operations of the general form:

base number$^{\text{exponent}}$

A familiar example of exponentiation is $5^2 = 25$.

The exponentiation operator used by the computer is the ∧ symbol, obtained by pressing the [∧] key ([SHIFT] 6).

Enter	Display
121∧.5	`11.`
6∧−3	`.0046296296`

The third precedence level is assigned to the unary operation "taking the negative of" as in the example

```
Enter     Display
x=9
-x        -9
```

This type of minus operation is called *unary*, meaning "one," to emphasize its difference from the operation of subtraction, which operates on two values.

The fourth precedence level is assigned to multiplication and division.

The fifth precedence level is assigned to addition and subtraction.

When the computer evaluates a calculation (hereafter referred to as a *numeric-expression*), it works through the expression from left to right performing: first, all higher-order functions in the expression; second, all exponentiations; third, all unary minuses; fourth, all multiplications and divisions; and finally, all additions and subtractions. When two or more operations of the same precedence are present, the left-most operation is performed first.

This order of precedence is followed whether an expression is evaluated in command mode or within a program.

The computer's capability to recognize and perform mathematical operations in the order expected by mathematicians is a great convenience. Sometimes it is necessary, however, to evaluate an expression in a different order.

If you want an expression to be evaluated in another order, put parentheses around the part of the expression you want evaluated separately. By enclosing a portion of the expression in parentheses, you instruct the computer to evaluate the operations within the parentheses before any operations outside of the parentheses. For example, entering

```
Enter           Display
2∧(3+10/2)      256
```

forces the computer to perform the division and addition operations before the exponentiation operation. Observe that within the parentheses, the normal rules of precedence still apply.

A calculation can require parentheses to be evaluated correctly even though it is not written with parentheses. For example, you will get the wrong answer if you enter

$$\frac{41+3}{17-6}$$

as 41 + 3 / 17 − 6.

In examples of this type, the numerator and denominator must be evaluated before the division operation is performed. Therefore, the correct way to enter this problem is

Enter	**Display**
(41+3)/(17−6)	4

If you are not certain that the computer will evaluate an expression in the order that you want, use parentheses. If necessary, use parentheses within parentheses. As long as you have defined the evaluation order correctly, the presence of extra parentheses will not disturb the calculation.

When parentheses are used within parentheses, the computer begins its evaluation process with the innermost set of parentheses and works outward.

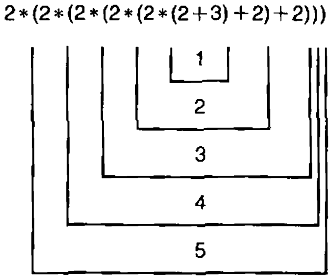


Fig. 11-1 Example illustrating the order of evaluation of parentheses.

Enter	Display
2*(2*(2*(2*(2*(2+3)+2)+2)))	208

The order in which this expression is evaluated is illustrated graphically in Fig. 11-1.

When using parentheses, match the pairs of parentheses carefully. If you enter an unequal number of left and right parentheses, the computer will display the error message `Unmatched parenthesis`.

LEARNED IN THIS LESSON

The order in which mathematical operations are evaluated is controlled by a ranking system usually referred to as the computer's *levels of precedence.* This system is based on conventions of mathematics and is designed to allow expressions that are entered as they are generally written to be evaluated in the correct algebraic order.

The levels of precedence used in evaluating mathematical operations are listed below.

Precedence	Operation Name	Operator Symbol
1	Higher-order functions	(see Lessons 12, 13, 14, and 25)
2	Exponentiation	∧
3	Unary minus (negation)	−
4	Multiplication and division	* and /
5	Addition and subtraction	+ and −

These operations are performed in a strict left-to-right order, beginning with the highest level of precedence. If two or more operations of the same level are present in an expression, the left-most operation is performed first.

Through the selective use of parentheses, the computer can be made to perform operations in any order. Parentheses instruct the computer to evaluate all operations within the parentheses before any operations outside of the parentheses. If you have any doubts about how the computer will evaluate an expression, use parentheses.

When parentheses are used within parentheses, the computer begins its evaluation process with the innermost set of parentheses and works outward.

Lesson 12

Higher-Order Mathematical Functions

The higher-order functions of the computer fall into two broad categories: mathematical and nonmathematical functions. This lesson briefly discusses the mathematical functions. Nonmathematical functions are discussed in Lessons 13, 14, and 25.

With one exception, the mathematical functions have the general form:

function(numeric-expression)

where *numeric-expression* represents the value operated upon by the function. This value, which can be a constant, variable, or calculation, is referred to as the *argument* of the function. The argument of a function must always be enclosed within parentheses or an error condition will occur.

Executing a function does not change the value of its argument. Thus,

```
display sqr(x)
```

displays the square root of x without changing the value of x. Because a function gives a result without changing its argument, it is said to *return* a value.

You can use a function in almost any way that you can use a variable. You can display it, manipulate it mathematically, base decision-making tests on it, or assign its value to a variable. You can even include it in the argument of another function. For example:

Enter	**Display**
display log(sqr(abs(−45)))	.8266062569

What you cannot do with a function is use it on the left side of an assignment sequence. Thus,

```
sqr(x)=11
```

is an illegal assignment sequence.

The mathematical function that does not take the general form shown above is **pi**, which returns the value of pi (π) to 13 significant digits. (Strictly speaking, **pi** is not really a function since it requires no argument and does not perform an operation. It is more accurately classified as a constant.)

The higher-order mathematical functions are listed below:

Function	**Definition**
sin(*numeric-expression***)**	sine
cos(*numeric-expression***)**	cosine
tan(*numeric-expression***)**	tangent
asn(*numeric-expression***)**	arcsine
acs(*numeric-expression***)**	arccosine
atn(*numeric-expression***)**	arctangent
log(*numeric-expression***)**	common logarithm
ln(*numeric-expression***)**	natural logarithm
exp(*numeric-expression***)**	natural antilogarithm
sqr(*numeric-expression***)**	square root
abs(*numeric-expression***)**	absolute value (makes a number positive)
pi	3.141592653590 (rounded to 3.141592654 in the display)

As you can see from the list, the name of a function is based upon what the function does. Although most of these names are distinct, you may find that **asn** and **acs** are similar enough to be confusing.

The trigonometric functions allow angular values to be entered or computed in degrees, radians, or grads, depending upon the angle-mode setting of the computer. The angle mode is selected by executing **deg**, **rad**, or **grad** in either command mode or within a program. The angle mode in effect at any time is indicated by the appearance of a DEG, RAD, or GRAD indicator in the display.

Several examples of mathematical functions follow:

Enter	**Display**
deg	
sin(30)	.5
log(pi)	.4971498727
sqr(121)	11
abs(−9.1)	9.1

LEARNED IN THIS LESSON

With the exception of **pi**, the higher-order mathematical functions have the general form:

function(*numeric-expression*)

where *numeric-expression* represents the *argument*, or value operated upon, of the function. The argument can be a constant, variable, or calculation, but must be enclosed in parentheses. Because a function gives a result without altering its argument, it is said to *return* a value.

The **pi** (π) function has no argument and always returns the value 3.14159265359.

The trigonometric functions allow angular values to be entered or computed in degrees, radians, or grads, depending upon the angle-mode setting of the computer. The angle mode is set by executing **deg, rad**, and **grad**, respectively. The angle mode currently in effect is shown by the appearance of a DEG, RAD, or GRAD indicator in the display.

Lesson 13

The INT and SGN Functions

The **int(***numeric-expression***)** function is a higher-order function used to find the integer portion of the value entered for numeric-expression. An *integer* is any of the natural numbers (1, 2, 3, etc.), the negatives of these numbers, or zero. For example, the statement

```
display int(7.459)
```

displays the natural number 7. For negative values, the **int** function returns a negative integer that is *less than or equal to the value of the argument.* Thus,

```
display int(−7.459)
```

displays −8 and not −7 as you might expect (−8 is less than −7.459; −7 is not).

The **int** function has a surprising variety of programming applications. It can be used, for example, to determine whether a value is even or odd by comparing the *integer result of dividing* the number by 2 with a *straightforward division* by 2. If the number is even, there is no fractional portion to discard after the division and the relation is true. If the number is odd, there is a fractional result left over and the relation is false.

```
new
100 pause all
110 input "Sample number? ";n
```

```
120 if int(n/2)=n/2 then display "The number is even"
130 if int(n/2)<>n/2 then display "The number is odd"
140 goto 110
```

A sample run of this program is illustrated below.

```
Enter      Display
run        Sample number?
99778      The number is even
[ENTER]    Sample number?
-333       The number is odd
[BREAK]    Break
```

This technique can be used any time you want to determine whether one number is evenly divisible by another. For example, if you are manipulating dates in a program, you may want to find out which years are leap years. The following program illustrates how **int** can make this check.

```
new
100 pause all
110 input "Sample year? ";year
120 if int(year/4)=year/4 then display "The year is a leap
    year"
130 if int(year/4)<>year/4 then display "The year is not a
    leap year"
140 goto 110
```

A sample **run** is shown below.

```
Enter      Display
run        Sample year?
1946       The year is not a leap year
[ENTER]    Sample year?
1428       The year is a leap year
[BREAK]    Break
```

The **int** function can also be used to round values to a desired number of decimal places. The general formula for this operation is

```
r=int(n*p+.5)/p
```

where r = the rounded value, n = the number to be rounded, and p = a power of ten such as 10, 100, 1000, or 10000. The value selected for p determines the number of places the value r is rounded to. For example, if p = 100, r is rounded to 2 decimal places. If p = 10000, r is rounded to 4 decimal places.

Briefly, the rounding formula works as follows: The value to be rounded is first multiplied by p to shift the decimal point a fixed number of places to the right. For example, if n is 12.3456, multiplying by 100 gives 1234.56. Next the equation adds .5 to this intermediate result. The purpose of this addition is to force decimal values of .5 to .9 to "round up" to the next higher unit. In the example quoted above, the intermediate result is now 1234.56 + .5 = 1235.06. Next the **int** function is used to discard any digits to the right of the decimal point. Finally, the intermediate result is divided by p to restore the decimal point to its original position. Thus, the final result in this example is 1235 / 100 = 12.35, or 12.3456 rounded to 2 decimal places.

(This formula is correct only for positive values. If a program must round negative values also, use the formula r=(**int**(**abs**(n)*p+.5)/p)***sgn**(n). See the end of the lesson for a discussion of the **sgn** function.)

The program shown below illustrates rounding numbers to 2 decimal places.

```
new
100 p=100
110 input "Sample number? ";n
120 r=int(n*p+.5)/p
130 display "The rounded number is";r
140 pause
150 goto 110
```

A sample run is shown below.

```
Enter      Display
run        Sample number?
1.495      The rounded number is 1.5
[ENTER]    Sample number?
```

```
1.494       The rounded number is 1.49
[BREAK]     Break
```

The program correctly rounds 1.495 up and 1.494 down.

If you want the program to round to another number of decimal places, change the value of p in line 100. For example, entering

```
100 p=100000
```

will cause the program to round to 5 decimal places..

The **int** function can also be used to find the fractional portion of a number. This is accomplished for positive numbers by subtracting the integer portion of the number from itself:

```
new
100 input "Sample number? ";n
110 f=n-int(n)
120 display "The fractional part is";f
130 pause
140 goto 100
```

A sample **run** is shown below.

```
Enter       Display
run         Sample number?
765.4321    The fractional part is .4321
[BREAK]     Break
```

Since the **int** function returns a value *less than or equal to the value of the argument* for negative arguments, this program will not work correctly for many negative numbers. For a method of handling negative numbers, see the discussion of the **sgn** function that follows.

Another higher-order function that is useful in testing the properties of a number is the **sgn(***numeric-expression***)** function. The **sgn** function returns a value of −1, 0, or 1 based upon the following rules:

Value of Argument	sgn Response
Less than 0	−1
Zero	0
Greater than 0	1

A program illustrating the operation of the **sgn** function is given below.

```
new
100 input "Sample number? ";n
110 display sgn(n)
120 pause .5
130 goto 100
```

A sample **run** is shown below.

```
Enter      Display
run        Sample number?
−7         -1
           Sample number?
0           0
           Sample number?
7           1
           Sample number?
BREAK      Break
```

The **sgn** function is often used to restore the sign of a number. For example, the value of the fractional portion of a negative number can be found using the formula

f=(**abs**(n)−**int**(**abs**(n)))∗**sgn**(n)

This formula works as follows. The absolute value of the fractional portion of the number is found, as indicated earlier, by subtracting the integer portion of the number from the number. The absolute value functions ensure that the correct fractional portion is found for both negative and positive numbers. After the fractional portion has been found, it is necessary to restore the minus sign for negative numbers. One way to do this would be to use an **if then** statement to test if the number was negative and then multiply by −1 when it was. It is more efficient to use the **sgn** function, however. Since **sgn** returns either −1, 0, or 1, multiplying by the value of **sgn**(n) will leave the sign unaltered when n is positive (or zero), and will change it to minus when n is negative.

The same technique can be applied to the rounding formula when it is necessary to handle negative numbers:

```
    r=(int(abs(n)*p+.5)/p)*sgn(n)
```

LEARNED IN THIS LESSON

The **int(***numeric-expression***)** function returns the integer value of numeric-expression. For positive arguments, this is the value of the argument stripped of any fractional or decimal part. For negative arguments, this is the negative integer just less than or equal to the value of the argument.

Common applications of the **int** function include determining if one number is divisible by another, rounding numbers to a fixed number of decimal places, and finding the fractional portion of a number.

The **sgn(***numeric-expression***)** function returns −1, 0, or 1, depending upon the sign of numeric-expression. If numeric-expression is negative, **sgn** returns −1. If numeric-expression equals zero, **sgn** returns 0. If numeric-expression is positive, **sgn** returns 1.

A common application of the **sgn** function is to restore the sign of a number after it has been removed.

Lesson 14

Simulating Chance Occurrences

Compact Computer BASIC has two higher-order functions which permit the computer to simulate chance or random occurrences. These functions create a series of numbers which appear to be random and have the same statistical properties as randomly generated numbers. Since these numbers are generated by a computational procedure, however, they are more accurately referred to as *pseudorandom numbers.*

The general form of the functions used to create pseudorandom numbers are given below. Notice that although **intrnd** requires an argument, **rnd** does not.

> **rnd**
>
> **intrnd(***numeric-expression***)**

The **rnd** function returns a number in the range from 0 through 1, where 0 is a possible value but 1 is not, each time the function is executed. A program illustrating **rnd** is shown below.

```
new
100 display rnd
110 pause
120 goto 100
```

Run the program.

Enter	**Display**
run	.8225408469
ENTER	.163164479
ENTER	.4352797351
ENTER	.3233738121
ENTER	.1733442624
BREAK	Break

While it appears that **rnd** does produce numbers in the range described, you are probably wondering how these numbers can be considered random. After all, didn't your program produce the same numbers as those printed in this book?

This result is not really contradictory and can be explained as follows. The computer begins its generation of pseudorandom numbers with an initial value known as a *seed value.* Unless you instruct it otherwise, the computer always uses the same seed value when it executes **rnd** for the first time in a program. Consequently, the same series of pseudorandom numbers is generated each time the program is run.

To instruct the computer to use a different seed value, you must place a **randomize** statement in your program before the first execution of **rnd**. The **randomize** statement instructs the computer to select an unpredictable seed value each time the program is run, which results in an unpredictable sequence of pseudorandom numbers.

Add the **randomize** statement to the program developed earlier by entering

```
    90 randomize
```

Notice that the **randomize** statement is given a line number outside of the loop used to generate and display the random numbers. The statement is located outside the loop because it is necessary to execute **randomize** only once in a program to get the desired unpredictable results.

Now **run** and break the program several times. You will get a different series of pseudorandom numbers each time you execute the program.

Technical Note: The reproducibility of the pseudorandom number sequence is actually an asset, as it allows complicated programs that use random numbers to be developed without worrying about whether operational differences are the result of programming errors or changes in the sequence of random numbers. Since **rnd** can be trusted to generate the same series of numbers when **randomize** is not used, any differences in results are attributable to programming problems. When a program is fully error free, the **randomize** statement can be added to create unpredictable random number sequences.

The following program illustrates how **rnd** can be put to a more practical use than displaying pseudorandom numbers. This program uses **rnd** to simulate the tossing of a coin.

```
new
100 randomize
110 r=rnd
120 if r<.5 then display at(1),"heads"
130 if r=>.5 then display at(7),"tails"
140 pause
150 goto 110
```

This program begins with **randomize** to ensure an unpredictable sequence of pseudorandom numbers. Then the program generates a random number. If the random number is less than .5, the message "heads" is displayed. If it is greater than or equal to .5, "tails" is displayed. (The nature of the pseudorandom number generating process of the computer is such that the likelihood of a pseudorandom number being in either of these categories is exactly equal.) The **pause** statement holds the appropriate message in the display until you press [ENTER], and the **goto** statement loops the program back to line 110 to repeat the sequence.

Run the program and observe the results. The sequence of heads and tails that are displayed should appear as random as actually flipping a coin.

Enter	**Display**
run	`heads`
[ENTER]	`      tails`

```
ENTER              tails
ENTER              tails
ENTER        heads
BREAK        Break
```

(The sequence shown by your computer will probably differ from the sample shown above.)

The **rnd** function is ideal for programs such as the heads or tails example where it doesn't matter that the random numbers are decimal values. For some applications, however, it is more convenient to have random numbers that are integers. The **intrnd(***numeric-expression***)** function should be used for these applications since it returns an integer value between 1 and the rounded value of numeric-expression. (The **randomize** statement must still be used to create an unpredictable sequence of numbers.)

The following program illustrates how **intrnd** can be used to simulate the roll of a pair of dice.

```
new
100 randomize
110 die1=intrnd(6)
120 die2=intrnd(6)
130 display die1,die2
140 pause
150 goto 110
```

The preceding programs illustrated how pseudorandom numbers can be used to simulate events that are truly random such as flipping a coin or rolling a pair of dice. Applications such as these are the basis for many computer games. Pseudorandom numbers are also useful for combining numeric or textual information in unpredictable ways. For example, a program that uses random numbers to provide drill practice for learning the multiplication tables is shown below.

```
new
100 randomize
110 n1=intrnd(9)
```

(*cont. on following page*)

(cont. from preceding page)

```
120 n2=intrnd(9)
130 display n1;"*";n2;"= ";
140 input answer
150 if answer=n1*n2 then 190
160 display beep,"WRONG. TRY AGAIN."
170 pause .8
180 goto 130
190 display "VERY GOOD!"
200 pause .8
210 goto 110
```

Observe that the pending display state created by line 130 prevents the **input** statement in line 140 from clearing the display.

A sample **run** of the program is shown below.

Enter	Display
run	4 * 6 = ?
24	VERY GOOD!
	8 * 7 = ?
48	WRONG. TRY AGAIN.
56	VERY GOOD!
	8 * 1 = ?
8	VERY GOOD!
	6 * 6 = ?
BREAK	Break

As written, this program provides drill practice for problems up to 9 × 9. The upper range of the program can be altered by changing the arguments of the **intrnd** functions. For example, using an argument of 11 provides practice up to 11 × 11.

You can alter the type of drill practice by changing the operation in line 150 and the prompt in line 130. For example, changing these lines to

```
130 display n1;"+";n2;"= ";
150 if answer=n1+n2 then 190
```

will generate addition problems.

LEARNED IN THIS LESSON

The two functions used to generate pseudorandom numbers are

rnd

intrnd(*numeric-expression***)**

The **rnd** function returns a pseudorandom number in the range 0 to 1, where 0 is a possible value but 1 is not. The **intrnd** function returns a pseudorandom integer number in the range 1 to the rounded value of numeric-expression, where both 1 and the rounded value of numeric-expression are possible values.

The **randomize** statement is used to ensure an unpredictable sequence of random numbers. If **randomize** is not used, the computer generates the same sequence of random numbers each time a program is run.

Review Test 3

1. What is meant by the phrase "levels of precedence"? What is the purpose of using parentheses in a calculation?

2. How can the exponentiation operation be used to compute the square root of a number? The fifth root of a number?

3. Which operation is performed first when you enter the sequence $-7*2$ into the computer?

4. Write one-line programs to display the results of the following problems:

   (a) $\sqrt{12+77} / \pi$

   (b) $118.56 \times 2.022^{2+3.8}$

   (c) $\frac{1}{9}+\frac{1}{7}+\frac{1}{5}+\frac{1}{3}$

   (d) $\dfrac{\frac{142}{117}+\frac{93}{12}}{\frac{16}{40}}$

5. Write a program to evaluate the following problem. Assume that 206 is in degrees, 94 is in grads, and 1.71 is in radians.

$$\frac{\sin 206 - \tan 94}{\cos 1.71}$$

6. What is the "argument" of a function? What is meant by stating that an argument can be any valid *numeric-expression*?

7. Write a program that computes the square root of any positive number that is input, but displays "INVALID" for any negative input. Use the **sgn** function to make the test.

8. What is the purpose of the **randomize** statement?

9. What do you think is the largest pseudorandom number generated by **rnd**? The smallest?

10. Using **rnd**, write a program to generate and sum 1000 random numbers. What is the sum likely to be?

11. Write a program that displays integer random numbers between 100 and 200.

Lesson 15

FOR TO NEXT Looping

In Lesson 10 you learned how to control the number of times that a loop was executed by establishing a "counting variable" and increasing it by 1 each time the loop was executed. An **if then** statement was used to determine when to stop executing the loop.

Because looping is such an important part of programming, BASIC provides two special statements for this purpose—the **for to** and **next** statements.

The **for to** and **next** statements have the general form:

> **for** *control-variable* = *initial-value* **to** *limit*
> •
> • (the statements to be executed in the loop go here)
> •
> **next** *control-variable*

where *control-variable* is the statement's "counting variable," *initial-value* is the starting value of the control-variable, and *limit* is the termination value. The instructions to be executed in the loop are placed between the **for to** and **next** statements.

The function of the **for to** statement is to specify the control-variable and define the starting and ending values of the variable. The **for to** statement also marks the physical beginning of the loop.

The function of the **next** statement is to increase the value of the control-variable and then decide if execution is to be sent back to the beginning of the loop or allowed to continue with the statement following the loop. If the control-variable is *less than or equal to* the limit value, execution is sent back to the first statement following the **for to** statement. If the control-variable is *greater than* the limit value, execution continues with the first statement following the **next** statement.

An example of a **for to next** loop is shown below.

```
new
100 for count=1 to 8
110 display count
120 pause .5
130 next count
140 display "loop done"
150 pause
```

Run the program and observe the results.

```
Enter      Display
run          1
             2
             3
             4
             5
             6
             7
             8
            loop done
```

The **for to** and **next** statements cause the computer to repeat the loop eight times.

When the **for to** statement is executed, the computer sets the value of count to 1. Execution then proceeds with lines 110 and 120. When the **next** statement is executed, the computer adds 1 to the value of count and checks whether it is time to end the loop. Since count is only equal to 2 at this point, execution is transferred back to line 110.

The program repeats this looping procedure seven more times. At the end of the eighth execution of the loop, the **next** statement increases the value of count to 9, which *exceeds* the limit specified by the **for to** statement. As a result the computer does not transfer execution back to line 110, but allows the program to continue with lines 140 and 150.

If you check the value of count after executing the program, you will find it to be 9.

Enter	**Display**
display count	9

The control-variable is larger than the limit.

The values selected for initial-value and limit determine the number of times the loop is executed. You can calculate this number using the formula

number of loops = limit + 1 − initial-value

The value of limit is increased by 1 because limit must be *exceeded* before the loop is exited.

You can use any values you want for initial-value and limit as long as they cause the computer to execute the loop the desired number of times. For example, change line 100 of this program to

```
100 for count=1001 to 1008
```

and **run** the program again.

Enter **Display**

```
run     1001
        1002
        1003
        1004
        1005
        1006
        1007
        1008
        loop done
```

The values displayed for count are different, of course, but the number of times that the loop is executed is the same (1008+1−1001=8).

As you can see, **for to next** looping is very similar to looping with a counting variable and an **if then** statement. The major difference is that the **next** statement combines the functions of increasing the counting variable, testing if the termination value has been reached, and transferring execution to the beginning of the loop if the termination value has not been reached. You may be wondering then, since they are so similar, which technique to use. Although there are instances in which a counting variable and an **if then** statement is the better choice, as a general rule you should use **for to** and **next** statements to create loops *when you know how many times the loop is to be executed.*

The next example illustrates how the averaging program developed in Lesson 10 can be modified to use a **for to next** loop.

```
new
100 input "Number of entries? ";n
110 for count=1 to n
120 input "Entry? ";entry
130 total=total+entry
140 next count
150 average=total/n
160 display "Average =";average
170 pause
```

In this program, a variable is used to specify the limit. You can use any valid numeric-expression as an initial-value or limit. If a variable is used, its value is not changed by the execution of the **for to** statement. Thus, you can be certain that n retains the correct value for calculating the average in line 150, even after it has been used to define the **for to next** limit.

Enter	**Display**
run	`Number of entries?`
5	`Entry?`

(*cont. on following page*)

(*cont. from preceding page*)

Enter	**Display**
11	`Entry?`
22	`Entry?`
33	`Entry?`
44	`Entry?`
55	`Average = 33`

The control-variable of a **for next** loop can do more than just count the number of times a loop is executed. When convenient, the control-variable can also be used in any calculations performed by the loop. If used in a calculation, however, it should not be changed unless you intend to change the number of times the loop is executed.

The following program provides a good example of an application where it is desirable to use a control-variable in a calculation. This program computes the factorial of any whole number that you enter up to 84 (the factorial of values larger than 84 exceed the computational limit of the computer). The factorial of a whole number is defined as the product of that number and all positive whole numbers less than the number. For example, the factorial of 7 = 1 × 2 × 3 × 4 × 5 × 6 × 7.

```
new
100 factorial=1
110 input "Enter number: ";n
120 if n>84 then 110
130 for count=1 to n
140 factorial=factorial*count
150 next count
160 display "The factorial is";factorial
170 pause
```

To compute the factorial, the program uses a loop to multiply the variable "factorial" by every whole number between 1 and the number entered.

A sample **run** of the program is shown below.

```
Enter     Display
run       Enter number:
12        The factorial is 479001600
[ENTER]
run       Enter number:
45        The factorial is 1.196222E+56
```

The factorial of 45 is too large to display in regular format, so it is displayed in scientific notation.

It is common in creating tables of numbers to use the control-variable of a loop in the calculation. The following program illustrates this technique.

```
new
100 onekarat=4.167
110 for karat=1 to 24
120 percentgold=karat*onekarat
130 display karat;"karats =";percentgold;"% gold"
140 pause .7
150 next karat
```

This program computes the percentage of gold in karat values 1 through 24. The calculations are performed by multiplying the value of the loop counter by 4.167 (the approximate percentage of gold in 1 karat). Since pure gold is 24 karats, the limit of the control-variable is 24.

A partial program **run** is shown below.

```
Enter    Display
run       1 karats = 4.167 % gold
          2 karats = 8.334 % gold
          3 karats = 12.501 % gold
          4 karats = 16.668 % gold
          •
          •
          •
          22 karats = 91.674 % gold
          23 karats = 95.841 % gold
          24 karats = 100.008 % gold
```

Here is another example where the control-variable of a **for ne** loop is used in a calculation. This program calculates the amoun of money generated by interest rates ranging from 10 to 15 percen compounded monthly for 12 months. The formula for this calculatio is $FV = P * (1 + i)^N$, where FV is the future value of the money P is the principal or amount deposited, i is the interest rate pe compounding period, and N is the number of compounding periods In this example, N equals 12 since the principal is compounde monthly for 1 year and i is the value of the control variable divide by 100 (the division by 100 is necessary to convert i to a decima value).

```
new
100 pause all
110 input "Principal: ";P
120 for count=10 to 15
130 i=count/100
140 FV=P*(1+i/12)∧12
150 FV=int(FV*100+.5)/100
160 display "Value at";count;"% =";FV;"dollars"
170 next count
```

The purpose of the calculation in line 150 is to round the value of FV to two decimal places. Refer to Lesson 13 for a discussion of this formula.

A sample **run** for a deposit of $1425.17 is shown below.

Enter	**Display**
run	`Principal`
1425.17	`Value at 10 % = 1574.4 dollars`
ENTER	`Value at 11 % = 1590.09 dollars`
ENTER	`Value at 12 % = 1605.92 dollars`
ENTER	`Value at 13 % = 1621.89 dollars`
ENTER	`Value at 14 % = 1638.01 dollars`
ENTER	`Value at 15 % = 1654.27 dollars`
ENTER	

LEARNED IN THIS LESSON

The function of the **for to** statement in a **for next** loop is to specify the counting variable of the loop (known as the *control-variable*) and define the starting and ending values of the variable, known as the *initial-value* and *limit,* respectively. The **for to** statement also marks the physical beginning of the loop. The value of the initial-value and limit can be defined by any valid numeric-expression.

The function of the **next** statement is to increase the value of the control-variable and decide if execution is to be sent back to the beginning of the loop or allowed to continue with the statement following the loop. If the control-variable is less than or equal to the limit-value, execution is sent back to the first statement following the **for to** statement. If the control-variable is greater than the limit value, execution continues with the first statement following the **next** statement.

The control-variable of a **for next** loop can be used in calculations within the loop when desirable. If used in a calculation, the control-variable should not be changed unless you intend to change the number of times the loop is executed.

It is recommended that you use **for next** loops whenever you know how many times the loop is to be executed. The advantages of **for next** loops are:

1. **For next** loops are shorter and more efficient than loops that use a counting variable and an **if then** statement.
2. **For next** loops are easier to write and reduce the chances of making a programming mistake.
3. A program that uses a **for next** loop is easier to understand than a program that uses an **if then** statement and a counting variable to control a loop.

Lesson 16

REM Statements

As you know by now, it is not always easy to understand how a program works. For this reason, BASIC provides a method of entering remarks or explanatory information directly into a program. The statement used for this purpose is the **rem** statement.

The **rem** statement instructs the computer to completely ignore the rest of the program line. As a result, you can place any information you want following a **rem** statement. You do not have to avoid using BASIC words or symbols in your remarks.

```
    new
    100 rem ********************
    110 rem This program consists
    120 rem entirely of rem statements.
    130 rem Since it contains
    140 rem no active statements,
    150 rem nothing happens
    160 rem when it is executed
    170 rem ********************
```

The information that you enter following a **rem** statement is unaltered. The computer does not convert lowercase letters to uppercase letters as it does for program statements.

It is good programming practice to use **rem** statements in your programs. Good remarks make a program much more understandable for others. They can also prevent you from wasting time trying to figure out what some part of your program does when you want to modify a program several weeks or months after writing it. Many programmers put their name, the name of the program, and the date that the program was written in **rem** statements at the beginning of every program they write. For longer programs, you may want to put information such as the variables used, the purpose of each variable, and the function of the major parts of the program in **rem** statements.

The following example illustrates the use of remarks in a program. Because of the many **rem** statements, you should need no additional explanation of the program's operation.

```
new
100 rem Balance checkbook program
110 rem n1=number of deposits
120 rem n2=number of withdrawals (checks)
130 input "Starting balance: ";balance
140 input "Number of deposits: ";n1
150 input "Number of withdrawals: ";n2
160 rem next line tests for no deposits
170 if n1=<0 then 230
180 rem ** begin deposit loop
190 for count=1 to n1
200 input "Deposit amount: ";deposit
210 balance=balance+deposit
220 next count
230 rem next line tests for no withdrawals
240 if n2=<0 then 300
250 rem ** begin withdrawal loop
260 for count=1 to n2
270 input "Withdrawal amount: ";withdrawal
280 balance=balance-withdrawal
290 next count
300 display "Your new balance is $";balance
310 pause
```

Notice that although **rem** statements do not perform any action during the running of a program, you can still transfer program execution to line numbers containing **rem** statements (line 170).

A sample **run** of the checkbook program is shown below.

```
Enter     Display
run       Starting balance:
412.19    Number of deposits:
1         Number of withdrawals:
5         Deposit amount:
10.42     Withdrawal amount:
125       Withdrawal amount:
79.51     Withdrawal amount:
9.95      Withdrawal amount:
12.98     Withdrawal amount:
5         Your new balance is $ 190.17
```

As an alternative to the word **rem**, you can use an exclamation point (!) to indicate a remark. The exclamation point serves the same function as **rem** and has the additional advantage that it can be placed at the end of a regular program line. An example of a program line that uses the exclamation point for a remark is shown below.

```
170 if n=<0 then 230!test for no deposits
```

LEARNED IN THIS LESSON

The function of the **rem** statement is to allow descriptive remarks to be placed within a program. The **rem** statement instructs the computer to ignore the remainder of the program line.

The exclamation point is an alternative to **rem** that can be used at the end of a regular program line.

Lesson 17

Storing DATA in a Program

Many programming applications require the manipulation of large quantities of information. For example, a payroll program might store the following information for each employee of a company: name, address, pay rate, marital status, and number of tax exemptions. This information *could* be placed in a program by using assignment statements, as shown below:

```
new
100 !payroll information
110 name1$="Brock"
120 address1$="7000 Semiconductor Drive"
130 rate1=6.03
140 status1$="married"
150 exemptions1=2
160 name2$="Clark"
170 address2$="4116 Memory Lane"
180 rate2=4.75
190 status2$="married"
200 exemptions2=2
•
•
•
```

But as you can see, if the company has many employees, this method of storing information requires a large number of variables and creates a program that is long and cumbersome. Fortunately, BASIC provides a more efficient method of storing information in a program. The basis of this alternative is the **data** statement.

The **data** statement has the general form

data *data-list*

where *data-list* is a list of string and/or numeric constants separated by commas.

The sole function of the **data** statement is to store information in a concise and organized manner. An example of the statement is shown below.

```
110 data Brock,7000 Semiconductor Drive,6.03,married,2
120 data Clark,4116 Memory Lane,5.32,married,2
```

Note that string information can be stored without quotation marks and that strings and numbers can be intermixed.

Technical Note: The use of quotation marks around string information in **data** statements is optional except when a string contains commas, leading or trailing spaces, or quotation marks. If you want to put commas or leading or trailing spaces in a string, you must enclose the string in quotation marks. If you want a string to contain quotation marks, enter *two* quotation marks where you want one quote to appear and enclose the string in quotation marks.

The information in all of the **data** statements of a program is considered to be one large list that begins with the first item of the first **data** statement and ends with the last item of the last **data** statement. Therefore, the number of items following a particular **data** statement is unimportant—what is important is the order in which items are listed. Thus,

```
110 data Brock,7000 Semiconductor Drive,6.03,married,2
```

and

```
110 data Brock
120 data 7000 Semiconductor Drive
130 data 6.03
```

```
140 data married
150 data 2
```

form identical lists of data as far as the computer is concerned. Of course, the first example stores the information in fewer program lines.

To access the information stored in a **data** statement, you must use the **read** statement. The general form of the **read** statement is

read *variable-list*

where *variable-list* is a list of string and/or numeric variables separated by commas.

The **read** statement assigns the information in a data-list to the variables in a variable-list. The data items are always assigned in the precise order in which they appear in the program. Thus the first **read** statement assigns the first data items to the variables in its variable-list, the second **read** statement assigns the next data items to the variables in its variable-list, and so on for each **read** statement that is executed. The computer keeps an exact count of the data items that have been assigned already and ensures that the next **read** statement always begins with the first unread data item.

The following example illustrates how this works.

```
new
100 !read data-list example
110 data Brock,7000 Semiconductor Drive,6.03,married,2
120 data Clark,4116 Memory Lane,5.32,married,2
130 data Dever,Software Circle,4.75,single,2
140 data Riddle,1024 Byte Road,4.15,married,2
150 data Smith,911 Silicon Park,7.75,single,1
160 data Stewart,8088 Processor Boulevard,6.95,single,1
170 for count=1 to 6
180 read name$,address$,rate,status$,taxexmpt
190 display name$,status$
200 pause 1
210 next count
```

A sample **run** of the program is shown below.

```
Enter    Display
run      Brock           married
         Clark           married
         Dever           single
         Riddle          married
         Smith           single
         Stewart         single
```

This program places the **read** statement in a loop and reads the data items into the same variable names each time the loop is executed. Although the information could be read into different variable names, one of the advantages of using **data** statements is that information does not have to be saved in variables since it has already been permanently stored in **data** statements. Observe that the program reads string information into string variables and numeric information into numeric variables. Compact Computer BASIC allows numeric information to be read into either a numeric or string variable, but it permits string information to be read only into a string variable.

Notice also that even though the information that is read into address$, rate, and taxexmpt is not used by the program, it is necessary to read that information in order to get the name and marital status of the next employee.

Data statements can be placed anywhere that you find convenient in a program. To verify this fact, delete lines 150 and 160 and reenter them at the end of the program.

```
del 150-160
220 data Smith,911 Silicon Park,7.75,single,1
230 data Stewart,8088 Processor Boulevard,6.95,single,1
```

The new location of the **data** statements does not alter the execution of the program.

```
Enter    Display
run      Brock           married
         Clark           married
```

(*cont. on following page*)

(cont. from preceding page)

```
Dever          single
Riddle         married
Smith          single
Stewart        single
```

Since **data** statements can appear anywhere in a program, different programmers develop different preferences for their location. Some prefer to place them at the beginning of a program, some at the end of a program, and some scatter them throughout a program.

When using **data** statements, you must be careful that your program does not attempt to read more data items than exist in the program. If a **read** statement is executed after the last data item has been read, a `DATA error` condition occurs.

To avoid data errors, you must prevent your program from reading past the last data item. This can often be accomplished by counting the number of data items and using that value in a **for next** loop. In some programming applications, however, you may want to add or subtract data each time the program is run. In such cases, a common technique is to store a "dummy" value as the last data item in the program. The program can then test the value of data items as they are read to determine when the end of the data-list is reached. The example that follows illustrates this programming technique.

```
new
100 !grocery list program
110 read desc$,quantity,unit$
120 if quantity=-1 then end
130 display quantity;unit$;" ";desc$
140 pause
150 goto 110
160 data eggs,2,dozen,milk,1,quart
170 data buns,3,packages,lettuce,2,heads
180 data softdrinks,7,liters
999 data xxx,-1,xxx
```

This program displays a list of foods to be bought on the next grocery trip. Since this list will presumably change as purchases are made or supplies are depleted, it is organized so that the last items in the data-list are dummy values. There must be three dummy items to correspond to the three variables in the **read** statement. The computer can test for a quantity of −1 to determine when the last data item has been read.

A sample **run** is shown below.

Enter	**Display**
run	`2 dozen eggs`
ENTER	`1 quart milk`
ENTER	`3 packages buns`
ENTER	`2 heads lettuce`
ENTER	`7 liters softdrinks`
ENTER	

As new supplies are needed, they can be added to the list by simply entering a new **data** statement with a line number less than 999. To remove a food from the list, delete the line containing the **data** statement where it is listed or remove it from the **data** statement. Just be certain to keep the string and numeric data in the correct order. The program will automatically adjust itself to the number of data items present.

```
del 160
190 data cheese,2,pounds
```

A sample **run** is shown below.

Enter	**Display**
run	`3 packages buns`
ENTER	`2 heads lettuce`
ENTER	`7 liters softdrinks`
ENTER	`2 pounds cheese`
ENTER	

It is often desirable to read a data-list more than once during the execution of a program. The **restore** statement allows a program to read a data-list over and over.

The general form of the **restore** statement is

restore *line number*

The **restore** statement specifies the line number where the next **read** statement that is executed is to begin reading data. The line number specified by the **restore** statement does not have to contain a **data** statement. If there is no **data** statement on that line, the **read** statement will begin with the first **data** statement after the specified line number. If no line number is given with the **restore** statement, the next **read** statement will begin with the first item in the data-list.

The following program illustrates the use of the **restore** statement.

```
new
100 !search for inventory item
110 pause all
120 input "Enter ID number: ";id
130 read n,quantity
140 if n=−1 then 180!test for end of data
150 if n<>id then 130!test for id number
160 display quantity;id;"in stock"
170 goto 190!skip next line
180 display id;"not found"
190 restore 100!reset data pointer
200 goto 120!loop
210 data 1001,7,1011,23,1003,3
220 data 1022,12,1009,1,1012,0
230 data 1033,4
999 data −1,−1!dummy data
```

The program searches through the information stored in a data-list for stock ID numbers. If the ID number is found, the quantity on hand is displayed. If the ID number is not found, the message "not found" is displayed. Since it is necessary to search through the data-list from the beginning each time a new ID is entered, a **restore** 100 statement is executed at the end of every search. The **restore** 100 ensures that the data-list is read from the beginning.

A sample **run** is shown below.

Enter	**Display**
run	`Enter ID number:`
1017	`1017 not found`
ENTER	`Enter ID number:`
1022	`12 1022 in stock`
BREAK	`Break`

LEARNED IN THIS LESSON

The **data** statement permits numeric and string information to be stored in a program without the requirement to assign it to variables. The items of information stored in a **data** statement must be separated by commas. It is not necessary to place quotation marks around strings unless the strings include commas, leading or trailing spaces, or quotation marks.

The **data** statement is not executed when a program is run. To access the information stored by **data** statements, it is necessary to use the **read** statement. The **read** statement assigns the information in **data** statements to variables in the **read** statement. **Data** statements can be placed wherever convenient in a program. It is not the location or number of **data** statements that is important, but the order in which information is listed in the statements. The **data** information is always assigned in the precise order in which it appears in the program.

Compact Computer BASIC allows numeric data to be read into either numeric or string variables, but it permits string data to be read only into string variables.

The **restore** statement can be used to allow a program to re-**read** data items. The general form of the statement is

restore *line number*

where *line number* indicates the location in the program where the next **read** statement is to begin reading. If the specified line number does not contain a **data** statement, reading begins with the first **data** statement found after that line number. If no line number is given, reading begins with the first item in the data-list.

Lesson 18

Arrays and Subscripts

The examples that you have studied so far have used what are known as *simple variables* to store information. Simple variables can be either numeric or string, but have the characteristic that they can be assigned only one value at a time. By contrast, an *array* variable can be assigned many values at a time. The different values assigned to an array variable are called *elements* and are kept separate by giving each value a unique *subscript.*

An example of an array variable is shown below.

```
        grade(3)=95
          ↗    ↖
     Array name    Subscript
```

This sequence assigns the value 95 to element 3 of an array named grade. The subscript that identifies the element is placed in parentheses immediately following the array's name.

Other array elements can be assigned just as easily.

```
grade(1)=88
grade(5)=100
```

The grade array as it is now defined is illustrated below. Notice that the first element is grade(0). *The first element of an array variable is always element 0.* Since you have not assigned values for elements 0, 2, and 4, they are shown as having zero values.

```
grade(0) = 0
grade(1) = 88
grade(2) = 0
grade(3) = 95
grade(4) = 0
grade(5) = 100
```

String arrays can be created by using a dollar sign as the last character of the array name. Thus,

```
name$(1)="Shakespeare"
```

assigns the string value "Shakespeare" to element 1 of a string array named name$.

Once a value has been assigned to a particular array element, you can use that element just as any variable as long as you specify the correct subscript when referring to it.

Enter	Display
grade(4)=52	
grade(4)*2	104
name$(0)="Hamlet"	
display name$(0)	Hamlet

It is the subscript following a variable name that identifies the variable as an array variable. If you drop the subscript, you are no longer referring to an array. Thus,

```
grade
grade2
grade(2)
```

are entirely different variables. The first two are simple variables and the third is an element of an array variable.

Technical Note: Although names such as grade and grade(2) refer to entirely different variables, you are not allowed to use identical names for an array and a simple variable in a program. If you do use the same name for an array and a simple variable, the error message `Variable previously defined` will be displayed when the second variable is encountered.

The rules for forming array names are the same as for forming simple variable names with the added restriction that the array element be specified by a subscript enclosed in parentheses. If the array is to contain string information, the last character of the array name must be a dollar sign. The following are examples of valid array names:

```
x(2)
@payroll(5)
item_cost(4)
day$(1)
L$(7)
```

The first three examples are numeric arrays. The last two examples are string arrays.

In the examples given so far, the number of the array element is specified by a constant. If you always had to use constants to specify an array element, array variables would be no more useful than simple variables. The strength of array variables is that subscripts can be specified by other variables. The following program provides an example of this feature.

```
new
100 !array demonstration program
110 for n=0 to 5
120 input "Enter sample number: ";sample(n)
130 next n
140 display "The numbers entered were: "
150 pause .5
160 for n=0 to 5
170 display "Element";n;"=";sample(n)
180 pause
190 next n
```

A sample **run** of the program appears below.

```
Enter      Display
run        Enter sample number:
100        Enter sample number:
```

(*cont. on following page*)

(*cont. from preceding page*)

```
Enter      Display
101        Enter sample number:
102        Enter sample number:
103        Enter sample number:
104        Enter sample number:
105        The numbers entered were:
           Element 0 = 100
[ENTER]    Element 1 = 101
[ENTER]    Element 2 = 102
[ENTER]    Element 3 = 103
[ENTER]    Element 4 = 104
[ENTER]    Element 5 = 105
```

This program accepts the numbers that you enter and stores them in elements 0 through 5 of an array named sample. After you have entered the six numbers, the program displays them one at a time, showing the array element in which they were stored.

The use of an array variable in this example makes it possible to enter and display the sample numbers using a loop, since the array element in which the numbers are stored is changed by simply changing the value of the subscript variable. If simple numeric variables were used instead of an array variable, then the input part of the program, for example, would require a sequence such as:

```
110 input "Enter sample number:" ;n0
120 input "Enter sample number:" ;n1
130 input "Enter sample number:" ;n2
140 input "Enter sample number:" ;n3
150 input "Enter sample number:" ;n4
160 input "Enter sample number:" ;n5
```

The display portion of the program would be equally repetitious.

If you modify the array demonstration program to accept 12 or more sample numbers, the error message `Bad subscript` will be displayed when the program attempts to assign a value to the twelfth element. The reason for this error is that the computer does not allow you to use an array with more than 11 elements unless you have first defined the size of the array.

The size of an array is defined by the **dim** statement. The **dim** (for *dim*ension) statement has the general form:

dim *array-name*(*integer-constant*)

where *array-name* is the name of the array variable and *integer-constant* is the number of the largest element of the array that will be used. *A variable or calculation cannot be used to specify the array size.* Since the first element in any array is element 0, the number of elements defined by the **dim** statement will be the value of integer-constant + 1.

Several examples of valid **dim** statements are shown below.

Statement	**Meaning**
dim test$(25)	25 is the largest valid element
dim miles(55)	55 is the largest valid element
dim n(12),p$(3)	12 is the largest valid element of array n, 3 is the largest valid element of array p$

Observe that a single **dim** statement can define the size of more than one array variable, as long as the different array names are separated by commas.

Although the computer will allow array variables of 11 or fewer elements to be used without requiring a **dim** statement, it is a good idea to define even small arrays. If you do not define an array variable's size with a **dim** statement, the computer will automatically reserve enough memory for 11 elements when it encounters the first array element. If you know that fewer than 11 elements are to be used, you can save memory by dimensioning the array for the smaller number of elements.

The following rules must be followed when dimensioning arrays:

- The **dim** statement must always appear on a lower-numbered line than that of the first occurrence of the array variable.

- An array can be dimensioned only once during the executio of a program. Thus, the size of an array cannot be change by a program after it has been defined. If you attempt t redimension an array, the error message `Variable pre viously defined` will be displayed.
- A **dim** statement cannot be used as the *action* of an I **then** statement.

An example of a program requiring a **dim** statement is shown below This program prompts for the entry of 12 monthly utility bills and stores the values that are entered in an array named bill. After all of the values are entered, the program displays the yearly total, the monthly average, the month of the highest charge, and the month of the lowest charge.

```
new
100 !monthly utility bill program
110 dim month$(12),bill(12)
120 for c=1 to 12
130 read month$(c)
140 input "Enter "&month$(c)&"'s bill: ";bill(c)
150 next c
160 pause all
170 total=0
180 smallest=1
190 largest=1
200 for c=1 to 12
210 total=total+bill(c)
220 if bill(c)<bill(smallest) then smallest=c
230 if bill(c)>bill(largest) then largest=c
240 next c
250 display "Yearly total =";total
260 display "Average bill =";total/12
270 display "Smallest bill was in ";month$(smallest)
280 display "Largest bill was in ";month$(largest)
290 data January,February,March,April
300 data May,June,July,August
310 data September,October,November, December
```

A sample program **run** is shown below.

```
Enter      Display
run        Enter January's bill:
46.15      Enter February's bill:
45.4       Enter March's bill:
40.88      Enter April's bill:
38.45      Enter May's bill:
30.07      Enter June's bill:
31.3       Enter July's bill:
39.52      Enter August's bill:
43.17      Enter September's bill:
36         Enter October's bill:
31.62      Enter November's bill:
35.22      Enter December's bill:
38.9       Yearly total = 456.68
ENTER      Average bill = 38.05666667
ENTER      Smallest bill was in May
ENTER      Largest bill was in January
```

Since the program uses two arrays larger than 11 elements, a **dim** statement is needed to define the arrays' sizes. In this case, both arrays are dimensioned to 13 elements (0–12), although element 0 is never used. After dimensioning the arrays, the program uses a loop to read the months of the year into the month$ array, and to input the monthly charges into the bill array.

The program sequence beginning with line 160 is responsible for finding and displaying the yearly total, the average, the month of lowest charge, and the month of highest charge. The sequence begins by setting total equal to 0, and setting smallest and largest to 1. The variables smallest and largest will be used to store the *number of the array element* with the smallest and largest values.

The program then enters the loop which computes the total and finds the two charges. The method used to find the total is a simple adding of all of the array elements. The method used to find the smallest and largest charges is described below.

1. When the loop begins, it is assumed that array element 1 contains the smallest and largest value. This provides a starting point for the first comparison.
2. On each execution of the loop, the program compares the value of the current array element (c) with the value of the array element it has previously found to be the smallest. If the current array element is found to be smaller, the value of *smallest* is set equal to the number of the current array element. On the next execution of the loop, the comparison will be with the new smallest element.
3. An identical technique is used to find the largest array element.
4. When the loop is complete, *smallest* will equal the number of the array element with the smallest charge, and *largest* will equal the number of the element with the largest charge. (If there are two identical smallest or largest charges, the program will find only the first.)

The program then displays the total, average, and month of the smallest and largest charges. Since the elements of the two arrays have a one-to-one correspondence, month$(smallest) will contain the name of the month with the smallest charge, and month$(largest) will contain the name of the month with the largest charge.

The preceding examples have illustrated what are known as *one-dimensional arrays.* A one-dimensional array is essentially a numbered list, where the number of the item in the list corresponds to the number of the array element. It is also possible to have *two-* (and *three-*) *dimensional arrays.*

A two-dimensional array can be visualized as a table of values, as illustrated in Fig. 18-1. Any element in a two-dimensional array can be identified by a row number and a column number. For example, the value in row 2, column 1 is 8.

As with one-dimensional arrays, the size of a two-dimensional array is defined with a **dim** statement. The general form of a two-dimensional **dim** statement is

> **dim** *array-name*(*row-integer*,*column-integer*)

	column 0	column 1	column 2	column 3
row 0	5	9	1	3
row 1	7	4	3	0
row 2	2	8	6	5

Fig. 18-1 An example of a table of values illustrating how an element of a two-dimensional array is defined by a row value and a column value.

where *row-integer* is the number of the largest row and *column-integer* is the number of the largest column. The row value is always specified before the column value. You are not required to use a **dim** statement for an array that has 11 or fewer rows and 11 or fewer columns.

A **dim** statement to define an array suitable for storing the table of values given in Fig. 18-1 is shown below.

```
dim table(2,3)
```

Notice that this statement defines an array that is 3 rows by 4 columns in size.

A program that allows you to enter values into an array that is 4 rows by 2 columns in size is shown below. The program then allows you to display the value of any element in the array.

```
new
100 !2 dimensional example
110 dim table(3,1)
120 for row=0 to 3
130 input "Sample number: ";table(row,0)
140 input "Sample number: ";table(row,1)
150 next row
160 input "Enter row value: ";row
170 input "Enter column value: ";column
180 display "Element";row;",";column;"=";table(row,column)
190 pause
200 goto 160
```

A sample **run** is given below.

```
Enter     Display
run       Sample number:
111       Sample number:
888       Sample number:
454       Sample number:
333       Sample number:
222       Sample number:
999       Sample number:
101       Sample number:
717       Enter row value:
2         Enter column value:
0         Element 2 , 0 = 222
[ENTER]   Enter row value:
0         Enter column value:
0         Element 0 , 0 = 111
[BREAK]   Break
```

Since the table array has been defined as having only 4 rows and 2 columns, the error message `Bad subscript` will be displayed if you enter a row value larger than 3 or a column value larger than 1. Entering a negative value will also cause the error to occur.

LEARNED IN THIS LESSON

Array variables have the capability to store many different values under one variable name. The different values assigned to an array variable are called *elements*. Each element is identified by a unique *subscript*.

The first element of any array is element 0.

The rules for forming array names are identical to those for forming simple variable names with the added restriction that the array element be specified by a subscript enclosed in parentheses immediately following the array name. If the array is to contain string information, the last character of the array name must be a dollar sign.

The **dim** (for *dim*ension) statement is used to define the size of an array. The general form of the statement for a one-dimensional array is

dim *array-name*(*integer-constant*)

where *array-name* is the name of the array variable and *integer-constant* is the number of the largest element that will be used. Since the first element in an array is element 0, the total number of elements defined by a **dim** statement is given by the value of integer-constant + 1. A single **dim** statement can dimension more than one array if the different array names are separated by commas.

The following rules must be followed when dimensioning arrays:

- The **dim** statement must always appear on a lower-numbered line than the first occurrence of an array variable.
- An array can be dimensioned only once in a program.
- A **dim** statement cannot be used as the *action* of an **if then** statement.

You are not required to have a **dim** statement for one-dimensional arrays with 11 or fewer elements. Nevertheless, it is a good idea to dimension even these small arrays, since dimensioning for less than 11 elements saves memory over not dimensioning at all.

Two-dimensional and three-dimensional arrays can also be created by the **dim** statement. The general form of a two-dimensional **dim** statement is

dim *array-name*(*row-integer*,*column-integer*)

where *row-integer* is the number of the largest row and *column-integer* is the number of the largest column. The row value is always specified before the column value. As with one-dimensional arrays, two-dimensional arrays of 11 or fewer rows and 11 or fewer columns do not have to be dimensioned, although defining the smaller arrays can save memory.

Lesson 19

The GOSUB and RETURN Statements

As you develop programs that are longer and more complicated, you will find that it is often necessary to repeat the same series of actions at different locations in a program. The most obvious way to write such a program is to reenter the necessary program lines each time they are needed. Of course, this method of handling repetition increases the size of the program and makes it more difficult and time-consuming to write.

Fortunately BASIC provides an important alternative known as a *subroutine.* A subroutine is a program segment that can be executed whenever needed by means of a **gosub** statement. The **gosub** statement has the general form

gosub *line number*

and instructs the computer to save a *return address* and then transfer program execution to the line number specified by the **gosub** statement. The return address consists of the location of the first program statement following the **gosub** statement.

Every subroutine must end with a **return** statement. When the **return** statement of a subroutine is executed, the computer sends program execution back to the *address that was saved when the gosub statement was executed.*

A program that illustrates the operation of the **gosub** and **return** statements by displaying a message indicating what part of the program is being executed is shown below.

```
new
100 !subroutine demonstration program             ┐
110 pause all                                     │
120 display "Executing main program"              │
130 gosub 180                                     ├─ Main program
140 display "Executing main program again"        │
150 gosub 180                                     │
160 display "Main program yet again"              │
170 end                                           ┘
180 !beginning of subroutine                      ┐
190 display "Now executing subroutine"            ├── Subroutine
200 return                                        ┘
```

A sample **run** of the program follows.

```
Enter      Display
run        Executing main program
[ENTER]    Now executing subroutine
[ENTER]    Executing main program again
[ENTER]    Now executing subroutine
[ENTER]    Main program yet again
```

When the computer executes the **gosub** 180 statement, it transfers execution to line 180, just as if a **goto** statement were executed. Since the **gosub** statement saves a return address, however, the **return** statement in the subroutine allows execution to resume with the appropriate main program line number. If a **goto** statement had been used instead of the **gosub** statement, no return address would be stored and the program would not know which line number to return to. Thus, the power of a subroutine is that it can be executed at any place in the main program *without permanently changing the order of program execution.*

To emphasize that a **gosub** statement only temporarily transfers execution to another location, programmers generally refer to the process of executing a subroutine as *"calling a subroutine."* The word *call* is understood by programmers to indicate that execution will return to the main program when the subroutine is complete.

Notice the **end** statement in line 170 of this program. This statem serves a very important purpose. If you remove the **end** statemen the computer will allow execution to continue into the subroutin after line 160 is executed. The first line of the subroutine will execut properly since it is a display statement, but when the comput attempts to execute the **return** statement, the error message RE-TURN without GOSUB will be displayed. It is always a mistak to allow a subroutine to be executed by a method other than **gosub** statement, since no return address is created for the **return** statement to use.

The computer uses a "last address in–first address out" method of keeping track of return addresses. Each time a **gosub** statement is executed, a return address is stored. Each time a **return** statement is executed, the return address stored last is erased from memory and used as the transfer address. This arrangement makes it possible for one subroutine to call another subroutine, a technique known as "nesting" subroutines.

To illustrate nesting subroutines, add the following lines to the subroutine demonstration program.

```
195 gosub 210
210 !beginning of second subroutine
220 display "Executing subroutine 2"
230 return
```

Run the program.

Enter	**Display**
run	Executing main program
ENTER	Now executing subroutine
ENTER	Executing subroutine 2
ENTER	Executing main program again
ENTER	Now executing subroutine
ENTER	Executing subroutine 2
ENTER	Main program yet again

Now, when the main program calls the subroutine at line 180, that subroutine displays its message and in turn calls a second subroutine at line 210. The second subroutine displays its message and then returns execution to line 200 of the first subroutine, which sends execution back to the appropriate line in the main program. Thus, program execution returns in the opposite order in which the subroutines are called.

It is also possible for the main program to call subroutine 2 directly. Make the following change to line 130 to demonstrate this feature.

```
130 gosub 210
```

A sample **run** appears below.

```
Enter      Display
run        Executing main program
[ENTER]    Executing subroutine 2
[ENTER]    Executing main program again
[ENTER]    Now executing subroutine
[ENTER]    Executing subroutine 2
[ENTER]    Main program yet again
```

The next program provides a more practical example of the use of subroutines. Before entering this program, however, be certain that you understand how the **gosub** and **return** statements affect the order of program execution. If you are unsure about some aspect of their operation, experiment with the subroutine demonstration program by adding subroutines and subroutine calls of your own. For example, you might want to add a third subroutine and have it called by subroutine 2. By writing a subroutine that displays a message identifying itself, the flow of the program is easy to follow.

The program that follows calculates the depreciation of an asset using the Accelerated Cost Recovery System (ACRS) adopted by the tax law of 1981. An asset life of 3 years is assumed. The function of the subroutine in the program is to calculate the amount of depreciation for a year and then round that value to two decimal places using the rounding formula discussed in Lesson 13. If a subroutine had not been used, it would be necessary to duplicate the statements that perform these calculations in three places in the program.

```
new
100 !program to calculate ACRS depreciation  ┐
110 pause all                                │
120 input "Enter cost of asset: ";cost       │
130 factor=.25                               │
140 gosub 240                                │
150 display "Depreciation year 1 =";depr     │
160 factor=.38                               ├─Main progr
170 gosub 240                                │
180 display "Depreciation year 2 =";depr     │
190 factor=.37                               │
200 gosub 240                                │
210 display "Depreciation year 3 =";depr     │
220 end                                      ┘
230 !depreciation subroutine                 ┐
240 depr=cost*factor                         ├── Subroutin
250 depr=int(depr*100+.5)/100                │
260 return                                   ┘
```

A sample **run** of the program is shown below.

Enter	**Display**
run	`Enter cost of asset:`
2995.95	`Depreciation year 1 = 748.99`
ENTER	`Depreciation year 2 = 1138.46`
ENTER	`Depreciation year 3 = 1108.5`

In addition to providing another example of the operation of the **gosub** and **return** statements, the ACRS program also illustrates how information can be passed between the main program and a subroutine. Just before calling the subroutine, the main program assigns the value to factor that the subroutine is to use in its calculations. The subroutine assigns the result of its calculations to the variable depr, which is then displayed by the main program. Thus, the fact that the values of variables are not changed by the flow of program execution is used to send and get information from the subroutine.

Here is a much longer subroutine example.

```
new
100 !Cards program
110 dim card$(52)
120 randomize
130 gosub 350!read cards into array
140 display at (1),"Shuffling deck . . ."
150 gosub 180!shuffle deck
160 gosub 270!deal cards
170 goto 140!repeat loop
180 !shuffle deck subroutine
190 for count=1 to 50
200 x=intrnd(52)
210 y=intrnd(52)
220 temp$=card$(x)
230 card$(x)=card$(y)
240 card$(y)=temp$
250 next count
260 return
270 !deal cards subroutine
280 s=intrnd(48)
290 for count=s to s+4
300 display card$(count);" ";
310 next count
320 display beep
330 pause
340 return
350 !read cards subroutine
360 for count=1 to 52
370 read card$(count)
380 next count
390 return
400 data 2S,3S,4S,5S,6S,7S,8S,9S,10S,JS,QS,KS,AS
410 data 2H,3H,4H,5H,6H,7H,8H,9H,10H,JH,QH,KH,AH
420 data 2D,3D,4D,5D,6D,7D,8D,9D,10D,JD,QD,KD,AD
430 data 2C,3C,4C,5C,6C,7C,8C,9C,10C,JC,QC,KC,AC
```

The function of this program is to create a deck of cards, shuffle the cards, and then deal out a hand of five cards. A sample **run** of the program is given below. Since the **randomize** function used, the cards that you see in your display and those shown this text will differ.

Enter	**Display**
run	`Shuffling deck...`
	`10C JH 8H 6S AH`
ENTER	`Shuffling deck...`
	`8S 10S 10C 3D 3H`
BREAK	`Break`

The abbreviations used to represent the suits are S for spades, H for hearts, D for diamonds, and C for clubs. The abbreviations for the nonnumeric card values are J for jack, Q for queen, K for king and A for ace.

The first point to observe about the card program is that the subroutines are not used to perform a function needed several times by the main program. Then why use them? The answer is that the subroutines split the program into units that are easier to understand. In fact, many programmers consider the organizational value of subroutines to be as important as the capability to reduce repetition. By separating the overall task of a program into smaller tasks and writing subroutines to perform those pieces, you can reduce complicated programs to manageable portions.

Briefly, here is how the card program works. The program begins by defining an array named card$ to hold the 52 cards. next, **randomize** is used to ensure a different seed number each time the program is executed. A subroutine is then called to **read** the card values from the **data** statements into the card$ array. This needs to be done only once each time the program is executed.

The subroutine beginning at line 200 is used to shuffle the deck of cards. The technique adopted for this process is to select two cards at random and then exchange their positions in the card$ array. The cards to be exchanged are selected by randomly assigning values between 1 and 52 to the variables x and y. Then, while the original value of card$(x) is temporarily stored in the variable temp$, the value of card$(y) is put into card$(x). To complete the exchange, the value of temp$ is assigned to card$(x). To mix the deck thoroughly, it is necessary to repeat this process a number of times. The **for next** loop in the subroutine repeats this sequence 50 times.

After the cards are shuffled, the subroutine beginning at line 300 is called to deal the hand. This subroutine begins by generating a random starting value between 1 and 48 so that the dealing begins at a different location each time (this is similar to "cutting" the deck). The upper value of this variable is placed at 48 so that the program will never attempt to deal past card$(52). The **for next** loop that follows displays five cards beginning with the randomly generated starting value. A semicolon is placed after the **display** statement to create a pending display state so that all of the cards can be seen at one time.

When execution returns from the dealing subroutine, the **goto** statement in line 170 loops the program back to the shuffling operation.

LEARNED IN THIS LESSON

Subroutines are executed by means of the **gosub** statement. The **gosub** statement instructs the computer to save a return address and then transfer execution to the line number in the **gosub** statement. Every subroutine must end with a **return** statement. The purpose of the **return** statement is to transfer program execution back to the return address created by the **gosub** statement.

To indicate that executing a **gosub** statement only temporarily sends execution to another location, programmers generally refer to the process of executing a subroutine as "calling a subroutine."

The "last address in–first address out" method of keeping track of subroutine addresses allows subroutines to call other subroutines, a technique known as "nesting" subroutines. When subroutines are nested, the flow of execution always returns in the opposite order in which the subroutines were executed.

Besides reducing the occurrence of repetition in a program, subroutines are an organizational tool. They allow large programming applications to be broken into smaller tasks that are more easily written and understood.

Review Test 4

1. Which of the following are valid BASIC statements?

   (a) 135 for f=T+7 to A*L
   (b) 200 data 126;17;5;15;7
   (c) 212 for f=−10 to 20
   (d) 116 read a,a,a,a
   (e) 999 data 1,6,3/2,16
   (f) 180 read a+2,a+3,a+4
   (g) 110 for k=0 to 0
   (h) 101 dim A(5.2)
   (i) 712 restore 100,200
   (j) 100 dim 3D(17)
   (k) 211 word(3)="test"
   (l) 500 dim swat(0)
   (m) 300 for a(2)=1 to 10
   (n) 500 for a=a(1) to 10
   (o) 222 dim a(g+7)
   (p) 100 dim week$(7),ID(12,7)
   (q) 950 grade(2)=grade2+grade3
   (r) 951 dim cost(32),cost(3,7)
   (s) 953 if employees=50 then dim wages(49)

2. Write a program that uses a **for next** loop to sum the whole numbers between 1 and 200, inclusive (1 + 2 + 3 + . . . 200).

3. A very rich man agrees to the following terms: for 31 days he will double the amount of money that he paid on the previous day. He begins by paying $.01 on day 1. Write a program that displays the following information for every day in the 31-day

period: the number of the day (beginning with 1 for the first day), the amount to be paid that day, and the total amount paid, including the amount to be paid that day.

4. What are the two methods of placing comments in a program? How do they differ?

5. What is the purpose of dummy values in **data** statements?

6. What is the purpose of the **restore** statement?

7. What is an array element? Does the variable pair$(5,6) describe one or two array elements?

8. What is the purpose of a **dim** statement? When is it required?

9. To dimension the A array for 20 elements, would you use A(19) or A(20)?

10. How many array elements does the following statement establish?

```
130 dim test(3,7,4)
```

11. What is the function of a **return** statement?

12. How are subroutines helpful in improving the organization of a program?

13. When is an **end** statement required in a program?

14. How is information passed to a subroutine and back to the main program?

Lesson 20

More on DISPLAYing Information

Lesson 3 introduced the **display** statement and illustrated its use. In addition to the **at** and **beep** options discussed in that lesson, the display statement allows the use of three other optional parameters: **erase all**, **size**, and **using**.

The **erase all** parameter instructs the computer to clear the contents of the display buffer before displaying any information. Since the computer normally clears the display buffer when it executes a **display** statement, an **erase all** parameter is needed only when you want to cancel the effects of a *pending display state.* As described in Lesson 8, a pending display state prevents the display buffer from being cleared when a **display** statement is executed. (A pending display state is created by putting a comma or semicolon after the last item of information in a print-list.) An example of the **erase all** option appears below.

```
200 display erase all beep,fee
```

Another **display** option is the **size(***numeric-expression***)** parameter. The **size** parameter allows you to limit the maximum number of characters displayed by the **display** statement to the value contained in parentheses following the **size** parameter. An example of the **size** option is given below.

```
125 display at(4) size(12),"Cost =";cost
```

The **size**(12) parameter limits the maximum number of characters that can be displayed to 12. Since the constant "Cost =" takes 6 characters, the size of the value of the variable cost is limited to a maximum of 6 characters, including the leading space or minus sign preceding the number.

The most powerful of the **display** options discussed in this lesson is the **using** parameter. When adopted, the **using** parameter must be the last parameter in the **display** statement. The **using** parameter instructs the computer to follow a specific format when displaying information. This format can be specified by either an **image** statement or a *format string.*

If an **image** statement is used, the number of the program line containing the **image** statement must follow the **using** parameter. If a format string is used, then that string must follow the **using** parameter.

The characters used to define the format are:

#	Specifies the position of a digit when a number is displayed or a character when a string is displayed. The amount of # signs used in the format establishes the maximum number of digits or characters that can be displayed.
.	Specifies the position of the decimal point in a number format. If omitted, no digits are displayed to the right of the decimal point.
^^^^	The use of either four or five of these ^ signs in a number format specifies that a number is to be displayed in scientific notation. For a discussion of scientific notation, refer to the *User's Guide* supplied with your computer.

An example that uses an **image** statement is shown below.

```
new
100 !example of image statement
110 input "Enter a sample number: ";n
120 display using 150,n
130 pause
```

(*cont. on following page*)

```
(cont. from preceding page)
140 goto 110
150 image ######.##
```

A sample **run** appears below.

```
Enter      Display
run        Enter a sample number:
-23           -23.00
[ENTER]    Enter a sample number:
.3456             .35
[ENTER]    Enter a sample number:
362432     362432.00
[BREAK]    Break
```

Observe that the computer fits the numbers into the format given by the **image** statement. Since the **image** statement specifies two digits to the right of the decimal point, the computer either displays zeros when there are no decimal digits in the number being displayed, or rounds to two decimal places when the number has more than two decimal digits. Notice also that while unused format positions to the left of the decimal point are displayed as spaces, the "sign" space that is ordinarily displayed in front of every positive number is omitted.

The six # signs to the left of the decimal point limit the number of digits that can be displayed there. If you attempt to display a number with more than six digits to the left of the decimal, the computer displays asterisks (*) for each format character in the **image** statement to indicate that something unexpected has occurred.

```
Enter      Display
run        Enter a sample number:
1234567    *********
[ENTER]    Enter a sample number:
-123456    *********
[BREAK]    Break
```

The minus sign is counted as a character, so the **image** statement also rejects −123456 as too large.

An **image** statement can contain information other than format characters. To illustrate this, replace line 150 with

```
150 image The number is: ######.##
```

and **run** the program again.

```
Enter        Display
run          Enter a sample number:
59.999       The number is:      60.00
[BREAK]      Break
```

Nonformat characters that are used in an **image** statement are referred to as *literals,* because they are displayed exactly as you enter them. You can use any character that you want as a literal except for a quotation mark. If you want a quotation mark to be displayed, then you must enclose all information within the **image** statement in quotation marks and put two quotation marks where you want the quotes to appear. For example, the statement

```
150 image "The ""number"" is: ######.##"
```

would display quotation marks around the word "number."

The **image** statement can also be used when displaying string information. To illustrate this possibility, make the following changes to the demonstration program.

```
110 input "Enter a sample string: ";s$
120 display using 150,s$
150 image The string is: ########
```

A sample **run** is shown below.

```
Enter          Display
run            Enter a sample string:
Dallas         The string is: Dallas
[ENTER]        Enter a sample string:
Albuquerque    The string is: ********
[BREAK]        Break
```

Again, the amount of # signs determines the maximum number of characters that can be displayed. Albuquerque is too long for this format, so the computer displays asterisks (*) when you attempt to display it. To avoid this result, anticipate the maximum number of characters that will be displayed and enter that many # signs.

An **image** statement can specify a format for more than one variable at a time. The following program illustrates this feature.

```
new
100 !grocery list program
110 read desc$,quantity,unit$
120 if quantity=-1 then end
130 display using 999,quantity;unit$;desc$
140 pause
150 goto 110
160 data eggs,2,dozen,milk,1,quart
170 data buns,3,packages,lettuce,2,heads
180 data drinks,7,liters
998 data xxx,-1,xxx!dummy values
999 image buy # ######## ########
```

A sample **run** is shown below.

```
Enter     Display
run       buy  2  dozen     eggs
[ENTER]   buy  1  quart     milk
[ENTER]   buy  3  packages  buns
[ENTER]   buy  2  heads     lettuce
[ENTER]   buy  7  liters    drinks
[ENTER]
```

The computer uses the three groups of format characters in the **image** statement to format the three variables.

As an alternative to an **image** statement, the **using** parameter can specify a format string. The format string, which can be either a variable or constant, uses the same format characters as an **image** statement.

You can modify the grocery list program to use a format string by deleting the **image** statement and substituting the following for line 130.

```
del 999
130 display using "buy # ######## ########",
    quantity;unit$;desc$
```

Notice that when a string constant is used, it must be enclosed in quotation marks. A format string can also be specified by a string variable or string expression, as illustrated below.

```
130 f$="buy # ######## ########"
135 display using f$,quantity;unit$;desc$
```

LEARNED IN THIS LESSON

In addition to the **at** and **beep** options, the **display** statement allows **erase all**, **size**, and **using** parameters. The **erase all** option clears the display buffer before displaying information, a capability that is useful for cancelling a pending display state. The **size** option limits the maximum number of numeric and string characters that can be displayed to the value specified in parentheses following the option. The **using** option specifies a format to be used for displaying information. When adopted, the **using** parameter must always be the last parameter in a **display** statement.

The two forms of the **using** parameter are

using *line number*

using *string-expression*

The **using** *line number* form instructs the computer to format information using an **image** statement. The **using** *string-expression* form instructs the computer to format information using the string variable or constant that follows the **using** parameter.

The characters used to define a display format are:

#	Specifies the position of a digit or string character.
	Specifies the position of the decimal point in a number format.
^^^^	Specifies the position of an exponent of a number to be displayed in scientific notation.

Any characters in a format other than those listed above are referred to as *literals* and are displayed "as is." The amount of # signs used in the format establishes the maximum number of digits or characters that can be displayed.

The **image** statement is similar to a **data** statement in that it is not executed directly. It is used only when referred to by a **using** parameter. Therefore, it can be placed anywhere convenient in a program, although as a rule **image** statements are located at the end of a program.

lesson 21

Multiple Statement Lines

In the examples you have seen so far, each program statement begins with a new line number. It is possible, however, to put more than one program statement on a line. Multiple statements can be placed on the same line by entering a colon between the statements. Except when used with an **if then** statement as explained later in this lesson, a colon instructs the computer to treat the statement that follows as if it was an entirely separate program line.

To illustrate this feature, look at the following program.

```
100 pause all
110 input "Feet? ";feet
120 meters=feet*.3048
130 display "Meters =";meters
```

The purpose of this program is to convert feet to meters. Here is the same program as one multistatement line.

```
100 pause all:input "Feet? ";feet:meters=feet*.3048:
    display "Meters =";meters
```

To combine the program lines, simply substitute a colon for the number of the next line. As long as the line does not exceed the 80-character limit imposed by the display buffer size, you can continue to add new statements to a line by preceding them with a colon. (The example given above is 77 characters long.)

Technical Note: Compact Computer BASIC does not permit any statement except a **rem** (or !) statement to appear on the same line with a **data**, **image**, or **dim** statement.

As you compare the two versions of the English-to-metric conversion program, it is evident that the first is easier to understand. Then why ever use multistatement lines? The primary answer is to save memory space. The computer uses 4 bytes of memory to store a line number, but only 1 byte to store a colon separator. Thus, replacing three line numbers with colons reduces the amount of memory needed to store the English-to-metric program by 9 bytes. This is calculated as follows:

$$\begin{aligned} \text{memory for 3 line numbers} &= 3 \times 4 = 12 \text{ bytes} \\ \text{memory for 3 colons} &= 3 \times 1 = \underline{\ \ 3} \text{ bytes} \\ \text{savings} &= \ \ 9 \text{ bytes} \end{aligned}$$

When memory space is not critical, however, it is usually better to avoid using multistatement lines, since they increase the complexity of a program. It is not uncommon for programmers to introduce errors into a program as a result of trying to save memory by combining lines. The more densely packed lines seem to breed errors.

Also, be aware that some program lines cannot be correctly combined. For example, a program such as

```
100 x=100
110 display x
120 x=x+2
130 goto 110
```

cannot be combined into one line because the **goto** statement can only transfer execution to the beginning of a line—not to the middle as would be required if this program was entered as one multistatement line. You *can* reduce this program to two lines, however.

```
100 x=100
110 display x:x=x+2:goto 110
```

The computer generally treats what follows a colon statement separator as an entirely separate program line. The exception to this rule occurs when the colon separator is used with an **if then** statement. To understand why the colon separator is treated differently in this case, you need to remember that the general form of the if **then** statement is

if *condition* **then** *action*

If the *condition* is true, the *action* is executed. If it is false, the *action* is skipped. Therefore, if a second statement is added to the action by means of a colon separator, that statement will also execute *only* when the condition is true; it will be skipped when the condition is false. The two examples that follow should help clarify this point.

Example 1

```
100 pause all
110 input "Enter a number: ";n
120 if n<0 then display "Negative"
130 goto 110
```

Example 2

```
100 pause all
110 input "Enter a number: ";n
120 if n<0 then display "Negative":goto 110
```

At first glance, these programs may appear to be functionally identical. They are not, although the only difference is the use of the colon separator. In Example 1, the computer is in a true endless loop. No matter what numeric value you enter, the computer will always send execution back to line 110.

In Example 2, the computer is not in an endless loop. The program will stop when you enter a positive value. The reason for this difference is that the **goto** statement in Example 2 is executed only when the condition is true—in this case, only when a negative number is entered. When a zero or positive value is entered, both the **display** and **goto** statements are skipped.

The computer's treatment of the colon separator following an **if then** statement is a useful feature. It simplifies the process of making multiple operations conditional. For example, consider the following program.

```
100 randomize
110 n1=intrnd(9)
120 n2=intrnd(9)
130 display n1;"*";n2;"= ";
140 input answer
150 if answer=n1*n2 then 190
160 display beep, "WRONG. TRY AGAIN."
170 pause .8
180 goto 130
190 display "VERY GOOD!"
200 pause .8
210 goto 110
```

This is a copy of the multiplication tables program discussed in Lesson 14. The purpose of the **if then** statement in line 150 is to check the answer that is entered. If the answer is correct, the programmer wants to display VERY GOOD, pause for .8 seconds, and then send execution back to generate a new problem. Because the program does not use multistatement lines, it is necessary to transfer execution to another part of the program to perform these actions. A version of the program that uses a multistatement line is given below.

```
100 randomize
110 n1=intrnd(9)
120 n2=intrnd(9)
130 display n1;"*";n2;"= ";
140 input answer
150 if answer=n1*n2 then display "VERY GOOD!":
    pause .8:goto 110
160 display beep,"WRONG. TRY AGAIN."
170 pause .8
180 goto 130
```

The second version is not just shorter—it is also more understandable. The consequences of a true decision-making result are obvious.

In the original version, by contrast, it is necessary to examine several parts of the program to find the consequences of a true result.

LEARNED IN THIS LESSON

Multiple statements can be placed on the same program line by entering a colon between the statements. Except when used with **if then** statements, the colon instructs the computer to treat the statement following the colon as if it was an entirely separate program line. When a colon separator is used following an **if then** statement, the additional statement or statements are part of the action of the **if then** statement. This is a useful feature that simplifies the execution of multiple actions by **if then** statements.

The maximum line length is 80 characters.

Compact Computer BASIC does not permit any statement except a **rem** (or **!**) statement to appear on the same line with a **data**, **image**, or **dim** statement.

Combining program statements shortens programs and reduces the memory needed to store them. This reduction in size is usually gained at the expense of program clarity. Unless program size is crucial, it is generally better to avoid densely packed multistatement lines.

Lesson 22

More on Decision Making

The purpose of the **if then** statement in a program is to permit the computer to perform different actions depending upon the result of a decision-making test. If the test is true, the action following the **then** portion of the **if then** statement is executed. If the test is false, the action is skipped.

By using the **else** option with the **if then** statement, you can define a specific action to be performed when the result of the test is false. The general form of the **if then** statement with the **else** option is

if *condition* **then** *action1* **else** *action2*

The **if then else** combination instructs the computer to perform *action1* when the *condition* is true and *action2* when the *condition* is false. In other words, "do action1 if true, do action2 if false."

To illustrate the convenience of the **else** option, examine the following program.

```
100 pause all
110 input "Enter a number: ";n
120 if n<0 then display "Negative"
130 display "Positive"
140 goto 110
```

Here the programmer wants to indicate whether the entered number is negative or positive (zero is assumed to be positive). The logic used to perform this task is not correct, however. Although the program does perform as intended when a positive number is entered, it does not work correctly with negative numbers. When a negative number is entered, the program displays both messages. It executes the action following the **then** statement *and* the program line that follows.

To fix this program, you can replace line 120 with

```
120 if n<0 then display "Negative":goto 110
```

or you can use the **else** option. The program as it appears if you use the **else** option is shown below.

```
100 pause all
110 input "Enter a number: ";n
120 if n<0 then display "Negative" else display "Positive"
130 goto 110
```

Although both the corrected original and the **else** version perform the intended task, the **else** version is shorter and easier to understand.

When multiple statements are used following an **else** option, they are treated as part of the action to be performed when the test result is false. For example, in the sample program line

```
300 if n=38 then display "Done":end else read t:n=n+t:
    goto 100
```

action1 consists of

```
display "Done":end
```

and action2 of

```
read t:n=n+t:goto 100
```

Notice that although colons are needed to separate multiple statements, no colon is needed to separate the **else** option.

LEARNED IN THIS LESSON

The **else** option is used to define a specific action to be performed when the result of a decision-making test is false. The general form of the **if then** statement with the **else** option is

> **if** *condition* **then** *action1* **else** *action2*

Action1 and *action2* can consist of single statements or multiple statements joined by the colon separator.

The advantage of the **else** option is that it shortens a program and generally makes it easier to understand.

Lesson 23

More on FOR NEXT Looping

The purpose of **for to next** statements in a program is to control the number of times that a loop is executed. In the examples in which these statements have been used so far, only one **for next** loop has been active at a time. It is possible, however, to "nest" **for next** loops just as you can nest subroutines.

An example that demonstrates the operation of nested **for next** loops is shown in Fig. 23-1. As you can see from the boxes drawn

```
new
100 !nested for next demonstration
110 pause all
+---------------------------------------------------+
|   120 for o=1 to 3                                |
|   130 display "outer loop, o =";o                 |
|  +----------------------------------+             |
|  | 140 for i=1 to 2                 |             |
|  | 150 display at(3),"inner loop, i =";i |—Inner loop |—Outer loop
|  | 160 next i                       |             |
|  +----------------------------------+             |
|   170 next o                                      |
+---------------------------------------------------+
```

Fig. 23-1 A program demonstrating the operation of nested **for next** loops.

around the loops, the inner loop is completely within the outer loop. This is absolutely essential when nesting **for next** loops. If you attempt to put any portion of the inner loop outside the outer loop, an error condition will result.

A sample **run** of the program from Fig. 23-1 is shown below.

```
Enter       Display
run         outer loop, o = 1
[ENTER]       inner loop, i = 1
[ENTER]       inner loop, i = 2
[ENTER]     outer loop, o = 2
[ENTER]       inner loop, i = 1
[ENTER]       inner loop, i = 2
[ENTER]     outer loop, o = 3
[ENTER]       inner loop, i = 1
[ENTER]       inner loop, i = 2
[ENTER]
```

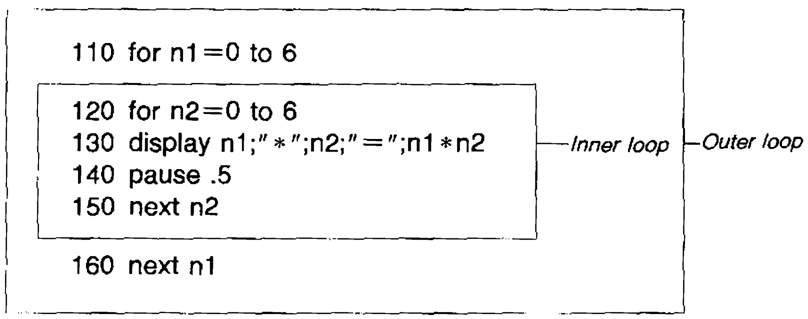
The program executes the entire inner loop every time it executes the outer loop once. Therefore, the inner control-variable always begins anew with the initial-value. While the inner loop is being executed, however, the computer keeps track of the current value of the control-variable of the outer loop. When the value of this variable exceeds 3, the program stops.

Compact Computer BASIC allows you to use as many as 24 nested loops in a program. When nesting loops, however, observe the following rules.

- Each loop must begin with its own **for to** statement and end with its own **next** statement.
- An inside loop must be located completely within the bounds of the next higher loop.
- Each loop must have a different control-variable.

Nested loops are very useful when you want to find all possible combinations, or permutations, of two or more variables. For example, suppose you want to display the multiplication table up to 6 × 6. The program shown in Fig. 23-2 demonstrates how this is done with nested **for next** loops.

```
new
100 !multiplication table program
```



```
110 for n1=0 to 6
120 for n2=0 to 6
130 display n1;"*";n2;"=";n1*n2
140 pause .5
150 next n2
160 next n1
```

Fig. 23-2 A program demonstrating how nested **for next** loops are used to find all permutations of the multiplication table up to 6 × 6.

A partial **run** of the program from Fig. 23-2 is shown below.

```
Enter     Display
run       0 * 0 = 0
          0 * 1 = 0
          0 * 2 = 0
          •
          •
          •
          6 * 5 = 30
          6 * 6 = 36
```

The first number in each of the multiplication problems is generated by the outer loop. The second number is generated by the inner loop. The two loops together generate all possible combinations of the values between 0 and 6.

The power and flexibility of **for next** loops can be greatly increased by using the **step** option. The **step** option allows you to define a specific increment to use for increasing the control-variable. The general form of the **for next** statement with the **step** option is

for *control-variable*=*initial-value* **to** *limit* **step** *increment*

The value given for *increment* can be either a constant or a variable.

The **step** option is useful primarily in cases where the control-variable is being used for several purposes. For example, in the compound interest program developed in Lesson 15, the value of the control-variable was also used as the annual interest rate. That program is reproduced below.

```
100 pause all
110 input "Principal: ";P
120 for count=10 to 15
130 i=count/100
140 FV=P*(1+i/12)∧12
150 FV=int(FV*100+.5)/100
160 display "Value at";count;"% =";FV;"dollars"
170 next count
```

By replacing line 120 of this program with

```
120 for count=10 to 15 step .5
```

you can calculate the effects of increasing the interest rate by .5 percentage points. The **step** .5 statement instructs the computer to increase the value of count by .5 rather than by 1. A sample **run** of the new version of the program is shown below.

Enter	**Display**
run	`Principal:`
987.23	`Value at 10 % = 1090.61 dollars`
$\boxed{\text{ENTER}}$	`Value at 10.5 % = 1096.03 dollars`
$\boxed{\text{ENTER}}$	`Value at 11 % = 1101.47 dollars`
$\boxed{\text{ENTER}}$	`Value at 11.5 % = 1106.94 dollars`
$\boxed{\text{ENTER}}$	`Value at 12 % = 1112.44 dollars`
$\boxed{\text{ENTER}}$	`Value at 12.5 % = 1117.96 dollars`
$\boxed{\text{ENTER}}$	`Value at 13 % = 1123.5 dollars`
$\boxed{\text{ENTER}}$	`Value at 13.5 % = 1129.07 dollars`
$\boxed{\text{ENTER}}$	`Value at 14 % = 1134.66 dollars`
$\boxed{\text{ENTER}}$	`Value at 14.5 % = 1140.29 dollars`
$\boxed{\text{ENTER}}$	`Value at 15 % = 1145.93 dollars`
$\boxed{\text{ENTER}}$	

By changing the increment of the **step** option, you can change the value by which the interest rate is increased. For example, to observe the effects of increasing the interest rate by 2 percentage points, replace line 120 with

```
120 for count=10 to 15 step 2
```

and **run** the program again.

```
Enter      Display
run        Principal:
987.23     Value at 10 % = 1090.61 dollars
ENTER      Value at 12 % = 1112.44 dollars
ENTER      Value at 14 % = 1134.66 dollars
ENTER
```

Since the control-variable is now being increased by two, the variable's limit is exceeded when 2 is added to 14.

Besides specifying a positive increment, the **step** option can specify a negative increment. If you then use the correct values for initial-value and limit, the **for next** loop can be made to count backwards. To illustrate this feature, make the following change to line 120.

```
120 for count=15 to 10 step -1
```

A sample **run** is shown below.

```
Enter      Display
run        Principal:
987.23     Value at 15 % = 1145.93 dollars
ENTER      Value at 14 % = 1134.66 dollars
ENTER      Value at 13 % = 1123.5 dollars
ENTER      Value at 12 % = 1112.44 dollars
ENTER      Value at 11 % = 1101.47 dollars
ENTER      Value at 10 % = 1090.61 dollars
ENTER
```

To make the **for next** loop count backwards, the initial value must be larger than the limit.

LEARNED IN THIS LESSON

For next loops can be nested by placing one loop within another loop. When nesting **for next** loops, the following rules must be observed.

- Each loop must begin with its own **for to** statement and end with its own **next** statement.
- An inner loop must be located completely within the bounds of an outer loop.
- Each loop must use a different control-variable.

The power and flexibility of **for next** loops can be increased by using the **step** option. The **step** option permits a **for next** loop to be incremented by any value, including negative or fractional values. The general form of the **for next** statement with the **step** option is

for *control-variable*=*initial-value* **to** *limit* **step** *increment*

The value given for the *increment* can be either a constant or a variable.

If a negative increment is used, and the initial-value is larger than the limit, the **for next** loop will count backwards.

Review Test 5

1. Which of the following are valid BASIC statements?

   (a) 222 display using 500 at(7),price
   (b) 100 display using #####.##,cost
   (c) 500 image "grand total = #####.##"
   (d) 250 display using S$&T$,h$
   (e) 190 display beep erase using 500,day$(x)
   (f) 200 if fee=1000 then fee=fee+15 goto 245
   (g) 110 gosub 200:rem call subroutine:x=x+1
   (h) 770 for x=1 to 10:display x;:next x
   (i) 135 read a,b,c:data 1,2,3
   (j) 400 if a=100 then 130 else if a=200 then 170
   (k) 450 for count=7 to 1 else −2
   (l) 300 for f=50 to −50 step −5
   (m) 235 for a=b to c step d
   (n) 266 for n1=19 to 9 step n:t=r/t:next n

2. What is the purpose of the **erase all** parameter in a **display** statement? When is it useful?

3. How is a display format specified for the **using** parameter? What symbols are used to specify the format?

4. How are multistatement lines created? When are they advantageous?

5. What is the function of the **else** option in an **if then** statement?

6. Write a program that uses the formula shown below to calculate and display the value of Y as X increases from 5 to 15 in increments of 0.5 units.

$$Y = 17X^2 + X - 2$$

7. A common application of nested **for next** loops is to read data into a two-dimensional array. Write a program that reads the following test scores from **data** statements into an array named grade and then calculates and displays the average test score for each student.

	Test 1	Test 2	Test 3	Average
Student 1	78	82	89	?
Student 2	83	85	81	?
Student 3	100	70	73	?
Student 4	65	68	71	?
Student 5	99	89	93	?
Student 6	87	0	93	?

Lesson 24

String Comparisons

Lesson 10 introduced the subject of relational tests and illustrated how they are used to compare numbers in decision-making instructions. Relational tests are not limited to numbers, however. Strings can be compared and tested just as easily.

A program that illustrates the use of string comparisons is shown below.

```
new
100 !string comparison example
110 pause all
120 input "Fahrenheit or Celsius? (F/C) ";a$
130 if a$="F" then gosub 160!calculate Fahrenheit
140 if a$="C" then gosub 220!calculate Celsius
150 goto 120!loop
160 !calculate Fahrenheit subroutine
170 input "Enter degrees Celsius: ";celsius
180 n=9/5*celsius+32
190 gosub 280!round to 2 places
200 display celsius;"Celsius =";n;"Fahrenheit"
210 return
220 !calculate Celsius subroutine
230 input "Enter degrees Fahrenheit: ";fahrenheit
240 n=5/9*(fahrenheit-32)
250 gosub 280!round to 2 places
```

(*cont. on following page*)

(cont. from preceding page)

```
260 display fahrenheit;"Fahrenheit =";n;"Celsius"
270 return
280 !rounding subroutine
290 n=int(n*100+.5)/100
300 return
```

Before running this program, look at lines 130 and 140. The purpose of these lines is to determine whether an "F" (for Fahrenheit) or a "C" (for Celsius) is entered in response to the "Fahrenheit or Celsius?" prompt. If an F is entered, the comparison at line 130 will be true and the subroutine at line 160 will be executed. If a C is entered, the comparison at line 140 will be true and the subroutine at line 220 will be executed. In either case, when execution returns from the subroutine, it proceeds to line 140 which sends the program back to line 120. If any character other than F or C is entered, both comparisons fail and again execution loops back to line 120.

A sample **run** is shown below.

```
Enter      Display
run        Fahrenheit or Celsius? (F/C)
C          Enter degrees Fahrenheit:
100         100 Fahrenheit = 37.78 Celsius
[ENTER]    Fahrenheit or Celsius? (F/C)
F          Enter degrees Celsius:
232.78      232.78 Celsius = 451 Fahrenheit
[BREAK]    Break
```

If you ran this program and it didn't accept your entry of an F or C, you were probably entering lowercase letters. Although it makes no difference whether you use lowercase or uppercase when entering BASIC commands or statements, the two cases are not considered equal by relational tests.

To understand how the computer distinguishes between uppercase and lowercase letters, you must know how string values are stored inside the computer. Internally, every string character is stored as a number between 0 and 255. These numbers are referred to as the ASCII *value* or *representation* of the character. ASCII is an acronym for American Standard Code for Information Interchange and defines a convention adopted by most computer manufacturers. A complete list of the ASCII values used to represent characters internally is given in Appendix A of this manual.

When the computer compares two string characters, it compares their ASCII values. If their ASCII values are identical, they are considered equal. Otherwise, they are considered unequal. In the case of f and F, the ASCII value of f is 102 and the ASCII value of F is 70. Therefore, they are not equal.

Because the computer uses ASCII values to make comparisons, it can also test whether a character is less than or greater than another character. For example, a lowercase f is greater than an uppercase F (102 > 70). A table showing the meaning of the relational operators when used to compare string characters is given below.

Symbol	**Meaning for String Comparisons**
<	Precedes in ASCII order
>	Follows in ASCII order
=	Equal to
<>	Not equal to
<=	Equals or precedes in ASCII order
>=	Equals or follows in ASCII order

Comparisons can be performed on strings longer than one character. To do so, the computer compares the strings character by character, including any spaces in the string. (If you look at the list in Appendix A, you will notice that the ASCII value of a "space" is 32. Therefore, a space is greater than any character with an ASCII value less than 32, and less than any character with an ASCII value greater than 32.)

The following program will help you understand string comparisons.

```
new
100 !string relation example
110 pause all
120 input "String 1: ";s1$
130 input "String 2: ";s2$
```

(*cont. on following page*)

(*cont. from preceding page*)

```
140 if s1$<s2$ then display s1$;"<";s2$
150 if s1$=s2$ then display s1$;"=";s2$
160 if s1$>s2$ then display s1$;">";s2$
170 goto 120
```

A sample **run** is shown below.

Enter	**Display**
run	String 1:
xyz	String 2:
XYZ	xyz>XYZ
ENTER	String 1:
$	String 2:
#	$>#
ENTER	String 1:
cat	String 2:
car	cat>car
ENTER	String 1:
Bill	String 2:
BILL	Bill>BILL
ENTER	String 1:
larger	String 2:
large	larger>large
ENTER	String 1:
isotherm	String 2:
isothermal	isotherm<isothermal
BREAK	Break

This program displays the relationship between two strings. You should be able to explain the results of the first four comparisons by referring to the ASCII table in Appendix A. To understand the last two comparisons, you must know the following rule: *if two strings of unequal length are equal up to where the shorter string ends, then the longer string is considered greater than the shorter string.* In effect, its extra length gives it a larger ASCII value. Therefore, "larger" is greater than "large", and "isothermal" is greater than "isotherm".

The null string, or string of zero length, is considered less than any other string:

Enter	**Display**
run	String 1:
ENTER	String 2:
.	<.
BREAK	Break

The null string is entered by just pressing ENTER. Since it contains nothing, nothing is shown to the left of the < sign.

You may be wondering when the null string is useful. The null string provides a convenient method of checking if an entry is made in response to an input prompt. Since it is easy to hit ENTER by mistake when responding to a prompt, it is a good idea to check for this error in any program that accepts the entry of string information. The next program provides an example of this type of error checking.

```
new
100 !telephone memo pad program
110 pause all
120 restore 200
130 input "Name: ";name$
140 if name$="" then 130
150 read a$,phone$
160 if a$="last name" then display "End of list":goto 120
170 if a$=name$ then display a$;"   ";phone$:goto 120
180 goto 150!loop
190 !start of data statements
200 data Steve,867-5309,Chris,521-0021
210 data Bob,522-2345,John,787-1964
220 data Nancy,212-322-7667,Ben,788-8828
230 data Ken,455-7223,Charles,787-9329 ext 231
999 data last name,last phone
```

This program is an electronic telephone directory. The names and numbers are stored in **data** statements, so new names and numbers can be added by simply entering **data** statements with line numbers less than 999. When the program is **run**, you are prompted to enter the name of the person whose phone number you want. The program searches through its data-list looking for the entered name.

If the name is found, the person's telephone number is displayed. Otherwise, "End of list" is displayed.

A sample **run** is shown below.

Enter	**Display**
run	`Name:`
Bob	`Bob 522-2345`
ENTER	`Name:`
charles	`End of list`
ENTER	`Name:`
Charles	`Charles 787-9329 ext 231`
BREAK	`Break`

Notice that the program requires that you enter the name you are looking for exactly as it is listed in the **data** statements.

LEARNED IN THIS LESSON

The computer allows all relational tests to be performed on string values. The tests are based upon a character by character comparison of the ASCII values of the strings. ASCII is an acronym for American Standard Code for Information Interchange and defines a convention adopted by most computer manufacturers for representing string characters in a computer's memory. A complete list of the ASCII values used by the CC-40 is given in Appendix A of this book.

A table showing the meaning of the relational operators when used to compare string characters is given below.

Symbol	**Meaning for String Comparisons**
<	Precedes in ASCII order
>	Follows in ASCII order
=	Equal to
<>	Not equal to
<=	Equals or precedes in ASCII order
>=	Equals or follows in ASCII order

The ASCII values for lowercase letters are greater than the values for the corresponding uppercase letters.

The null string, or string of zero length, is considered to be less than any other string.

Lesson 25

String Manipulations

In the telephone memo pad program developed in the preceding lesson, you had to enter an entire name, correctly spelled and capitalized, if you wanted the program to find a telephone number. This requirement not only increased the difficulty of using the program, it placed the burden of exactly remembering the person's name on your memory. By using the **seg$** function, you can place most of that burden on the computer's memory.

The **seg$** function allows a program to split a string into pieces, for either assignment or testing purposes. The general form of the **seg$** function is

> **seg$(***string-expression*,*position*,*length***)**

where *string-expression* is the string constant or variable you wish to split, *position* is the place in the string at which the segmenting is to begin, and *length* is the number of characters to split off. The following short program will help you understand how this function works.

```
new
100 !seg$ demonstration program
110 pause all
120 test$="abcdefghijklmnopqrstuvwxyz"
130 input "Enter starting position: ";start
140 input "Enter length: ";length
```

```
150 display seg$(test$,start,length)
160 goto 120
```

This program defines test$ as the lowercase letters of the alphabet from a to z. It then displays a segment of that string based upon the values you enter for starting position and length. A sample **run** is shown below.

Enter	**Display**
run	`Enter starting position:`
1	`Enter length:`
12	`abcdefghijkl`
ENTER	`Enter starting position:`
25	`Enter length:`
2	`yz`
ENTER	`Enter starting position:`
11	`Enter length:`
7	`klmnopq`
BREAK	`Break`

The first position in a string is 1, so if you enter a zero or negative number for starting position, a `Bad argument` error condition will result when the **seg$** function is executed. You can enter any nonnegative value that you want for the length. If you enter a value that is greater than the number of characters from the starting position to the end of the string, the computer simply takes the rest of the string.

By using **seg$**, you can modify the telephone memo pad program so that you only have to enter the first letter of a name and it will find all names beginning with that letter.

```
new
100 !telephone memo pad program
110 pause all
120 restore 200
130 input "Name: ";name$
140 if name$="" then 130
150 read a$,phone$
160 if a$="last name" then display "End of list":goto 120
```

(*cont. on following page*)

(*cont. from preceding page*)

```
170 if seg$(a$,1,1)=seg$(name$,1,1) then display a$;
    "  ";phone$
180 goto 150
190 !start of data statements
200 data Steve,867-5309,Chris,521-0021
210 data Bob,522-2345,John,787-1964
220 data Nancy,212-322-7667,Ben,788-8828
230 data Ken,455-7223,Charles,787-9329 ext 231
999 data last name,last phone
```

The only change made to the original version was in line 170. This change causes the program to compare a segment of a$ starting at position 1 with a length of 1 to a segment of name$ starting at position 1 with a length of 1 (in other words, the first letter of a$ with the first letter of name$). If the two are equal, a match has been found and the name and phone number are displayed. Since there can be many names with the same first letter, the program continues to search the **data** statements until the end of the list, marked by the dummy values "last name" and "last phone", is found.

A sample **run** is shown below.

```
Enter       Display
run         Name:
B           Bob  522-2345
[ENTER]     Ben  788-8828
[ENTER]     End of list
[ENTER]     Name:
C           Chris  521-0021
[ENTER]     Charles  787-9329 ext 231
[ENTER]     End of list
[BREAK]     Break
```

Since the program now looks only at the first letter of each string, it will display the names Chris and Charles even if you enter Cz for the name.

Another useful string function is the **len** function. The **len** function allows a program to determine how long a string is (i.e., how many characters are in the string). The general form of the **len** function is

len(*string-expression***)**

where *string-expression* can be a string constant, variable, or concatenated expression.

A program that demonstrates the **len** function is shown below.

```
new
100 !len demonstration program
110 input "Enter string: ";s$
120 display s$;" is";len(s$);"characters long"
130 pause
140 goto 110
```

A sample **run** is shown below.

```
Enter       Display
run         Enter string:
test it     test it is 7 characters long
[ENTER]     Enter string:
[ENTER]      is 0 characters long
[BREAK]     Break
```

The **len** function counts all characters in a string, including spaces. If a null string is entered, it has a length of zero.

The **rpt$** function creates a string of a specific number of characters. The general form of the function is

rpt$(*string-expression,numeric-expression***)**

where *string-expression* is the string to be repeated and *numeric-expression* is the number of times it is to be repeated.

The following example demonstrates the operation of the **rpt$** function.

```
new
100 !rpt$ demonstration program
110 input "Enter string: ";s$
120 display rpt$(s$,5)
```

(*cont. on following page*)

(*cont. from preceding page*)

```
130 pause
140 display "The string length is:";len(rpt$(s$,5))
150 pause
160 goto 110
```

A sample **run** is shown below.

```
Enter        Display
run          Enter string:
qwerty       qwertyqwertyqwertyqwertyqwerty
[ENTER]      The string length is: 30
[ENTER]      Enter string:
_            _ _ _ _ _
[ENTER]      The string length is: 5
[BREAK]      Break
```

The **rpt$** program displays five repetitions of the string that you enter. (Other numbers of repetitions can be obtained by changing the numeric-expression parameter.) When you press ENTER, the program then displays the length of the string. Observe that this program does not actually assign the result of the **rpt$** function to a string variable. When desirable, this can be done by an assignment statement such as

```
test$=rpt$(s$,5)
```

Notice also that a string function can be used as the argument of another string function (line 140). This type of construction is valid as long as the inner function produces the kind of result expected by the outer function. In this case, the **rpt$** function produces a string value, exactly as required by the **len** function.

The maximum string length allowed by the computer is 255 characters. If you attempt to produce a string that is longer than this, the computer will cut the string length back to 255 characters and display the message `String truncation`. (You can check this with the **rpt$** demonstration program above by entering a string that is more than 51 characters long.)

The next program puts the three string manipulation functions to work. This program provides a simple version of a word guessing game. The game requires two players. The first player enters a word while the second player is not looking. The second player must discover the word by guessing which letters are in the word. Every time a correct guess is made, that letter is put in the appropriate place(s) in the blank word. The computer keeps a count of how many guesses were made and displays that number at the end of each round.

```
new
100 !guess word program
110 pause all
120 input "Enter secret word: ";word$
130 if word$="" then 120
140 length=len(word$)
150 guesscount=0
160 letterscorrect=0
170 format$=rpt$("-",length)
180 !begin guessing loop
190 if letterscorrect=length then 350!round finished
200 display format$;"  ";guesscount;
210 input " Guess? ";guess$
220 if guess$="" then 210
230 guesscount=guesscount+1
240 !begin string search loop
250 for position=1 to length
260 if seg$(word$,position,1)=guess$ then gosub 290
270 next position
280 goto 180
290 !put letter in format$ subroutine
300 left$=seg$(format$,1,position-1)
310 right$=seg$(format$,position+1,length-position+1)
320 format$=left$&guess$&right$
330 letterscorrect=letterscorrect+1
340 return
350 !round finished
360 display "You guessed it in";guesscount;"guesses"
370 goto 120
```

A sample **run** is shown below.

Enter	**Display**
run	`Enter secret word:`
help	`---- 0 Guess?`
h	`h--- 1 Guess?`
z	`h--- 2 Guess?`
p	`h--p 3 Guess?`
e	`he-p 4 Guess?`
l	`You guessed it in 5 guesses`
BREAK	`Break`

A brief explanation of the program follows.

The program begins by prompting for the secret word and storing that word in word$. The **len** function is used to determine how long the secret word is. After setting the variables guesscount and letterscorrect to zero, the program uses the value produced by the **len** function to create a format string showing how long the secret word is. The **rpt$** function is used for this purpose since it allows a variable to set the string length.

Line 180 begins the guessing loop of the program. The loop begins by checking if the number of correct letters (letterscorrect) equals the number of letters in the secret word (length). If this test is true, the word has been guessed and the round is over.

If letterscorrect does not equal length, the program proceeds to prompt for a guess. The variable guesscount is increased each time a guess is made so that the program can display the number of guesses at the end of the round.

The string search part of the program is a loop that uses the **seg$** function to check each of the positions in the secret word to determine if a correct guess has been made. When a correct guess is found, the subroutine at line 290 puts the letter in the correct location in the format string, increases the value of letterscorrect by one, and returns execution to the string search loop. After every letter in the string has been checked, execution is transferred to line 180 to begin the guessing loop again.

LEARNED IN THIS LESSON

The **seg$** function is used to split a string into a segment or piece. The general form of the function is

seg$(*string-expression*,*position*,*length***)**

where *string-expression* is the string to be split, *position* is the place where the segmenting is to begin, and *length* is the number of characters to split off. The first character in the string is position 1.

The **len** function determines the number of characters in a string. The general form of the function is

len(*string-expression***)**

where *string-expression* is the string whose length is to be tested.

The **rpt$** function is used to create a string that consists of a certain number of repetitions of another string. The general form of the function is

rpt$(*string-expression*,*numeric-expression***)**

where *string-expression* is repeated the number of times specified by *numeric-expression.*

Lesson 26

Logical Operations

In a decision-making statement such as

```
if day>31 then gosub 200
```

the computer decides to execute or not execute the subroutine based upon the result of the relational test day>31. If the relation is true, the subroutine call is executed. If the relation is false, the subroutine call is skipped.

Decisions can also be based upon the results of logical operators. Logical operators allow several true-false conditions to be tested with a single **if then** statement. The logical operators are: **and**, **or**, **xor**, and **not**.

The **and** operator compares two separate true-false conditions and arrives at a single true or false result, based upon the rules shown in the table below. The results of the individual true-false tests (represented by X and Y) are shown on the left and the combined result is shown on the right.

X	Y	X and Y
True	True	True
True	False	False
False	True	False
False	False	False

As you can see from studying the table, for the result of an **and** operation to be true, both individual tests must be true. If either of the individual tests is false, the combined result is false.

An example of an **and** decision-making test is shown below.

```
    100 if month=4 and day=31 then display "Invalid day"
```

If month equals 4 and day equals 31, the message will be displayed. If either or both variables have some other value, the **display** statement will be skipped.

The rules controlling the operation of the **or** operator are shown below.

X	Y	X or Y
True	True	True
True	False	True
False	True	True
False	False	False

The result of an **or** operation is false only when both individual results are false. If either or both individual results are true, the combined result is true also. An example of an **or** test is shown below.

```
    100 if day<1 or day>31 then display "Invalid day"
```

If day is less than 1 or greater than 31, the message will be displayed.

The rules controlling the operation of the **xor** (exclusive or) operator are shown below.

X	Y	X xor Y
True	True	False
True	False	True
False	True	True
False	False	False

The result of an **xor** test is true only when the results of the individual tests differ. Whenever the individual tests have identical results, the combined result is false.

An example of an **xor** test is shown below.

```
100 if a=5 xor b=10 then gosub 200
```

If either a or b equals the tested value, but not both, the subroutine call is executed. If both are true, or both are false, the subroutine call is skipped.

The last logical operator is the **not** operator. The **not** operator reverses the result of a single true-false test.

X	not X
True	False
False	True

An example of the **not** operator is shown below.

```
100 if not a=5 then gosub 200
```

If a equals 5, the subroutine call will be skipped. If a does not equal 5, the subroutine call will be executed.

The logical operators can be used to create very sophisticated decision-making tests. For example:

```
if a=5 and b<10 or c>15 and not d=20 then x=2
```

When different logical operators are combined in one decision-making test, the operators are evaluated from left to right in the following order:

Levels of Precedence for Logical Operators	
not	(highest priority)
xor	
and	
or	(lowest priority)

The order in which the decision-making example given above will be evaluated is illustrated in Fig. 26-1. The first operator performed is the **not** operator; the last operator performed is the **or** operator.

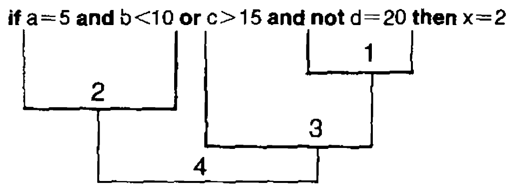


Fig. 26-1 Example illustrating the order of evaluation of multiple logical operators.

If you want logical operators to be evaluated in another order, you can place parentheses around portions of the test, just as in mathematical calculations. For example, entering

```
if a=5 and (b<10 or c>15) then x=2
```

will force the computer to evaluate the **or** operator before the **and** operator.

The major advantage of logical operators is that they allow programs to be shortened by combining several **if then** tests into one statement. The following program demonstrates how convenient this can be. The purpose of this program is to ensure that values entered for month, day, and year are valid. Without the capability to use logical operators in these tests, the program would require many more **if then** statements.

```
new
100 !validate date program
110 pause all
120 input "Enter month: ";m
130 if m<1 or m>12 or m<>int(m) then 220
140 input "Enter day: ";day
150 if day<1 or day>31 or day<>int(day) then 240
160 if day=31 and (m=4 or m=6 or m=9 or m=11) then
    240
170 if day>29 and m=2 then 240
180 input "Enter year: ";year
190 if year/4<>int(year/4) and m=2 and day>28 then 260
200 display m;day;year;"is a valid date"
210 goto 120!loop
220 display "Invalid month"
```

(*cont. on following page*)

```
(cont. from preceding page)
230 goto 120
240 display "Invalid day"
250 goto 140
260 display "Year is not a leap year"
270 display "so day is invalid"
280 goto 140
```

The functions of the decision-making tests are explained below.

Line 130:	Rejects the entered month if it is less than 1 or greater than 12 or a fractional value.
Line 150:	Rejects the entered day if it is less than 1 or greater than 31 or a fractional value.
Line 160:	Rejects the entered day if it is equal to 31 and any one of the following months was entered: April, June, September, or November.
Line 170:	Rejects the entered day if it is greater than 29 and the month is February.
Line 190:	Rejects the entered day if the year is not a leap year and the month is February and the day is equal to 29 (see Lesson 13 for a discussion of the leap year test).

A sample program **run** appears below.

```
Enter     Display
run       Enter month:
2         Enter day:
29        Enter year:
1981      Year is not a leap year
ENTER     so day is invalid
ENTER     Enter day:
29        Enter year:
1980       2  29  1980 is a valid date
ENTER     Enter month:
7         Enter day:
31        Enter year:
1949       7  31  1949 is a valid date
BREAK     Break
```

LEARNED IN THIS LESSON

Logical operators allow several true-false conditions to be tested with a single **if then** statement. The logical operators available in Compact Computer BASIC are **and**, **or**, **xor**, and **not**.

The **and**, **or**, and **xor** operators produce a single true or false result from a comparison of two separate decision-making tests. A table giving the rules governing these operations is shown below (X and Y represent the results of the individual tests).

X	Y	X and Y	X or Y	X xor Y
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

In summary, **and** is true when both individual tests are true, **or** is true when at least one individual test is true, and **xor** is true when the results of the individual tests differ.

The **not** operator reverses the status of a single decision-making test, as shown in the table below.

X	not X
True	False
False	True

Lesson 27
The ON GOTO and ON GOSUB Statements

The **on goto** and **on gosub** statements are decision-making statements that base decisions on a numeric value and not on the result of a true-false condition like the **if then** statement. These statements have the general form:

> **on** *numeric-expression* **goto** *line-list*
>
> **on** *numeric-expression* **gosub** *line-list*

where *numeric-expression* is a numeric variable or calculation and *line-list* is a list of line numbers separated by commas.

The value of numeric-expression determines which of the transfer locations in the line-list is selected. If numeric-expression equals 1, the program transfers to the first line number; if 2, to the second line number; if 3, to the third line number, and so on. If numeric-expression has a value that is less than 1, greater than the number of line numbers in the line-list, or is not an integer, a `Bad value` error condition occurs.

A sample **on goto** statement is shown below.

```
50 on n goto 100,200,300
```

When this statement is executed, the computer will transfer program execution to line 100 if n=1, to line 200 if n=2, and to line 300 if n=3. A single **on goto** statement such as this can replace a three-line program sequence such as

```
50 if n=1 then 100
60 if n=2 then 200
70 if n=3 then 300
```

The **on gosub** statement functions identically to the **on goto** statement except that the line numbers in the line-list are treated as subroutines rather than direct transfers. For example, when

```
250 on k/2+1 gosub 3000,205,3100,2000
```

is executed, the program calls line 3000 if k/2+1=1, line 205 if k/2+1=2, line 3100 if k/2+1=3, and line 2000 if k/2+1=4. When the program returns from the selected subroutine, execution continues with the first line following the **on gosub** statement.

A program that uses the **on goto** statement is shown below.

```
new
100 !Meters/liters/grams conversion program
110 display "1-Meters 2-Liters 3-Grams";
120 input option
130 if option<0 or option>3 then 120
140 on option goto 150,190,230
150 !Convert feet to meters
160 input "Enter feet: ";feet
170 display feet;"feet =";feet*.3048;"meters"
180 pause:goto 110
190 !Convert gallons to liters
200 input "Enter gallons: ";gallons
210 display gallons;"gallons =";gallons*3.785;"liters"
220 pause:goto 110
230 !Convert ounces to grams
240 input "Enter ounces: ";ounces
250 display ounces;"ounces =";ounces*28.349;"grams"
260 pause:goto 110
```

This program illustrates a common application of the **on goto** anc **on gosub** statements. The program begins by displaying a list, oı *menu,* of conversion options. To select one of the conversions you enter the number of that option. After checking that you have entered a valid option number, the program uses an **on goto** statement to direct execution to the correct conversion routine.

A sample program **run** appears below.

```
Enter     Display
run       1-Meters 2-Liters 3-Grams?
1         Enter feet:
45         45 feet = 13.716 meters
ENTER     1-Meters 2-Liters 3-Grams?
3         Enter ounces:
89.9       89.9 ounces = 2548.5751 grams
ENTER     1-Meters 2-Liters 3-Grams?
2         Enter gallons:
13.1       13.1 gallons = 49.5835 liters
BREAK     Break
```

The next example illustrates an application of the **on gosub** statement. This program calculates the day of the week for any date after October 15, 1582. (The program is not accurate for dates before October 15, 1582, the year the Gregorian calendar was instituted by Pope Gregory XIII.) The program uses a set of equations based upon a formula known as *Zeller's congruence.* The equations are:

(1) T=365*YEAR+DAY+31*(MONTH−1)

(2) *For January and February:*
F=T+int((YEAR−1)/4)−int(.75*(int(((YEAR−1)/100))+1))
For March through December:
F=T−int(.4*MONTH+2.3)+int(YEAR/4)
−int(.75*(int(YEAR/100)+1))

(3) W=F+(−1*(int(F/7)*7))

where MONTH is the numeric month (1–12), DAY the calendar day (1–31), and YEAR is the year (1582+) of the target date. T and F are used for temporary storage of intermediate results. The final result of the calculation is a factor W (0–6) that indicates the day of the week based upon the values shown below:

Value of W	Day of Week
0	Saturday
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday

Notice that Equation (2) is different depending upon whether the month falls in the range January–February or March–December.

The program is given below.

```
new
100 !Day of week program
110 input "Enter month: ";month
120 input "Enter day: ";day
130 input "Enter year: ";year
140 t=365*year+day+31*(month-1)!equation 1
150 if month<=2 then 240!check for Jan or Feb
160 !equation 2 for Mar-Dec
170 f=t-int(.4*month+2.3)+int(year/4)-int(.75*
    (int(year/100)+1))
180 !calculate day of week factor
190 w=f+(-1*(int(f/7)*7))!equation 3
200 on w+1 gosub 270,280,290,300,310,320,330
210 display "The day is: ";day$
220 pause
230 goto 110!loop
240 !equation 2 for Jan-Feb
250 f=t+int((year-1)/4)-int(.75*(int(((year-1)/100))+1))
260 goto 180!go calculate w
270 day$="Saturday":return
280 day$="Sunday":return
290 day$="Monday":return
300 day$="Tuesday":return
310 day$="Wednesday":return
```
(cont. on following page)

(*cont. from preceding page*)

```
320 day$="Thursday":return
330 day$="Friday":return
```

The **on gosub** statement is used to assign the correct day of the week to day$ after w has been calculated. (To keep the example short, no attempt is made to check that a valid date is entered. As a practice exercise, combine this program with the date checking program given in Lesson 26 to ensure that only valid dates are entered.)

A sample **run** is shown below.

```
Enter     Display
run       Enter month:
11        Enter day:
19        Enter year:
1946      The day is: Tuesday
[ENTER]   Enter month:
4         Enter day:
25        Enter year:
2001      The day is: Wednesday
[BREAK]   Break
```

LEARNED IN THIS LESSON

The **on goto** and **on gosub** statements are decision-making statements that differ from the **if then** statement in that

- They base decisions on a numeric value and not on the results of a true or false test.
- They allow a single statement to transfer program execution to more than one program location.

The **on goto** and **on gosub** statements have the general form

on *numeric-expression* **goto** *line-list*

on *numeric-expression* **gosub** *line-list*

where *numeric-expression* is a numeric variable or calculation and *line-list* is a list of line numbers separated by commas.

The computer uses a simple counting scheme to select the line number in the line-list. The program branches to the first line number if the value of numeric-expression is 1, to the second line number if it is 2, to the third line number if it is 3, and so on. If numeric-expression has a value that is less than 1 or greater than the number of transfer locations in the line-list, a `Bad value` error condition occurs.

Lesson 28

Subprograms

A *subprogram,* like a subroutine, is a program sequence designed to perform a specific function for a main program and then return program execution to the first instruction following the subprogram calling sequence. A subprogram is a more independent organizational unit than a subroutine, however. The variables used by the subprogram are independent of the variables used by the main program, even if the variable names are the same, and information to be sent to or received from a subprogram must be communicated in a special way.

The computer can execute four types of subprograms: user-written subprograms (your subprograms), built-in subprograms, machine language subprograms, and subprograms in *Solid State Software*™ cartridges installed in the cartridge port of the computer. This lesson discusses the first two subprogram types. For information on machine language or *Solid State Software* subprograms, refer to the *User's Guide* provided with your computer.

Subprograms are executed by the **call** statement. The **call** statement has the general form:

call *subprogram-name*(*argument-list*)

where *subprogram-name* is the name of the subprogram and *argument-list* is a list of constants, variables, array names, or calculations that are separated by commas. The elements in the argument-list of a **call** statement are called arguments. The argument-list parameter is optional and can be omitted if no information is to be exchanged between the main program and the subprogram.

Subprogram names can be almost any sequence of letters or digits up to 15 characters in length, as long as the first character of the name is a letter of the alphabet or underline character. A list of names that cannot be used because they are the names of built-in subprograms is given later in this lesson.

Subprograms must always follow the main program and any subroutines used by the main program, as illustrated graphically below.

```
Start of main program
•
•
•
End of main program
Start of subroutines used by main program
•
•
•
End of subroutines used by main program
Start of subprogram
•
•
•
End of subprogram.
```

Every subprogram must begin with a **sub** statement. The **sub** statement has the general form:

sub *subprogram-name*(*parameter-list*)

where *subprogram-name* must match the name that is used in the **call** statement to execute the subprogram. The *parameter-list* in the **sub** statement must meet the following criteria: It must have a numeric element where the argument-list of the corresponding **call** statement has a numeric element, and it must have a string element where the **call** statement has a string element. If the *argument-list* and the *parameter-list* do not match as described above, the error message `Illegal syntax` will be displayed when the **call** statement is executed.

Just as every subprogram must begin with a **sub** statement, it must end with a **subend** statement. The **subend** statement marks the end of the subprogram and returns execution to the first instruction following the **call** statement used to execute the subprogram.

A **subend** statement cannot be used as the conditional action of an **if then** statement. If you need to return execution to the main program as the result of a conditional test, you must use a **subexit** statement as shown below.

```
2000 if count=done then subexit
```

The last line of a subprogram must still have a **subend** statement, even though a **subexit** statement is used to return execution to the main program.

The following example illustrates how the **call**, **sub**, and **subend** statements are applied.

```
new
100 !subprogram example 1      ─┐
110 m$="The count is"           │
120 for count=1 to 100          ├── Main program
130 call display(m$,count)      │
140 next count                  │
150 end                        ─┘
160 sub display(t$,x)          ─┐
170 display t$;x                ├── Subprogram
180 subend                     ─┘
```

This example uses a subprogram to display the value of count. Of course, you would not normally use a subprogram to perform a task as simple as this. The program does illustrate several important points about using subprograms, however.

First, notice that the name of the subprogram is a reserved word (**display**). It is valid to use reserved words as subprogram names.

Second, notice the correspondence between the argument-list of the **call** statement and the parameter-list of the **sub** statement.

Both contain a string variable and a numeric variable, in that order. The **call** statement passes the values of m$ and count to the subprogram. The **sub** program links the value of m$ to t$, and the value of count to x. Consequently, if the subprogram changes the value of t$, the value of m$ changes; if the subprogram changes the value of x, the value of count changes.

A partial sample **run** is shown below.

```
Enter     Display
run       The count is 1
          The count is 2
          •
          •
          •
          The count is 99
          The count is 100
```

Another example of a program that uses a subprogram is shown below. In this example, information is passed to the subprogram, modified there, and then returned to the main program.

```
new
100 !subprogram example 2                    ┐
110 markup=.15                               │
120 input "Enter cost: ";cost                │
130 call calc(cost,markup)                   ├── Main program
140 display using 170,cost                   │
150 pause                                    │
160 goto 120                                 │
170 image The price is ####.##               ┘
180 sub calc(cost,factor)                    ┐
190 cost=cost+cost*factor                    ├── Subprogram
200 subend                                   ┘
```

In this example, a variable named "cost" is used by both the main program and the subprogram.

Technical Note: Observe that the **image** statement used by the main program is placed before the beginning of the subprogram.

Compact Computer BASIC requires that **image** and **data** statements intended for use by the main program be located within the main program, and **image** and **data** statements for use by a subprogram be located within the subprogram.

A sample **run** of example 2 appears below.

```
Enter      Display
run        Enter cost:
19.12      The price is   21.99
[ENTER]    Enter cost:
109.5      The price is  125.93
[BREAK]    Break
```

The subprogram calculates the marked-up price and passes it back to the main program, which displays it.

Subprogram examples 1 and 2 illustrate what is called "*passing arguments by reference*" by the *User's Guide* supplied with the CC-40. This phrase means that the subprogram *can change the value of a variable in the argument-list* of a **call** statement *by changing the corresponding variable in the parameter-list* of the subprogram (as in example 2). *The corresponding variables do not have to have the same name.* Arrays are always passed by reference.

The **call** statement can also "*pass arguments by value*." When arguments are passed by value, a subprogram *cannot change the value of a variable in the argument-list, even if the corresponding variable* in the **sub** statement is changed. Compact Computer BASIC requires that constants and calculations always be passed by value. Variables can be passed by value by enclosing them in parentheses within the argument-list. The following example illustrates passing information by value.

```
new
100 !subprogram example 3
110 markup=.15
120 input "Enter cost: ";cost
130 call calc(cost,(markup))
```

```
140 display using 170,cost
150 pause
160 goto 120
170 image The price is ####.##
180 sub calc(cost,factor)
190 cost=cost+cost*factor
200 factor=.55
210 subend
```

This program is a modification of the previous example. In this version, the amount of markup is being passed to the subprogram "by value," so it is enclosed in parentheses. As demonstrated by line 200, when an arugment is passed by value, it cannot be changed, even though the corresponding variable in the parameter-list is changed.

Technical Note: The major difference between passing an argument by reference and passing an argument by value is this: When an argument is passed *by reference,* the subprogram uses the same variable as the main program, even though it may have been given another name by the **sub** statement. Therefore, if the variable is changed by any statement in the subprogram, the new value will be used by the main program when execution returns to the main program. When an argument is passed *by value,* the subprogram establishes a new variable with a value equal to the corresponding argument of the argument-list. Thus, any changes to the new variable will not be passed back to the main program, even if the variable names are the same.

In addition to executing subprograms of your own creation, the **call** statement can execute subprograms built into the BASIC language of the computer. A list of the names of the built-in subprograms is given below. These names cannot be used as names of user-written subprograms.

addmem	err
char	exec
cleanup	getlang
debug	getmem

(*cont. on next page*)

```
indic        poke
io           relmem
key          setlang
load         version
peek
```

More detailed information on the functions of the built-in subprograms, including a description of the argument parameters required by the subprograms, is provided in the *User's Guide* supplied with the computer.

An example illustrating how a built-in subprogram is executed is shown below.

```
new
100 !call indic demonstration
110 x=3:y=4
120 for count=1 to 9
130 call indic(x,1):call indic(y,1)
140 pause .1
150 x=x-1:y=y+1:if x=-1 then x=17
160 next count
170 for count=1 to 9
180 call indic(x,0):call indic(y,0)
190 x=x-1:y=y+1:if y=18 then y=0
200 pause .1
210 next count
```

This program uses the **indic** subprogram to alternately turn the display indicators on and off. The **indic** subprogram requires two arguments, as shown below.

call indic(*indicator-number*,*indicator-state***)**

The first argument indicates which indicator is being referenced. An illustration showing the 18 display indicators and the numbers assigned to them is given in Fig. 28-1. The second argument indicates whether the subprogram is to turn the indicator on or off. If the second argument has a value of 0, the indicator is turned off; if it has any value other than 0, the indicator is turned on.

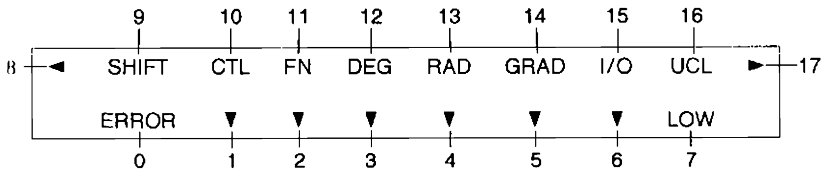


Fig. 28-1 Illustration showing the 18 display indicators and the numbers assigned to them.

LEARNED IN THIS LESSON

A subprogram is an independent program sequence designed to perform a specific function for a main program. A subprogram is executed by the **call** statement. The general form of the **call** statement is

call *subprogram-name*(*argument-list*)

The **call** statement specifies the name of the subprogram being executed and lists the information, or *arguments,* being passed to the subprogram.

A subprogram must begin with the **sub** statement. The general form of the **sub** statement is

sub *subprogram-name*(*parameter-list*)

The subprogram-name in the **call** statement must match exactly the name used by the **sub** statement. The argument-list must have a numeric element where the parameter-list has a numeric element, and a string element where the parameter-list has a string element.

A subprogram must end with a **subend** statement. The **subend** statement marks the physical end of the subprogram and returns execution to the first instruction following the **call** statement used to execute the subprogram.

A **subend** statement cannot be used as the conditional action of an **if then** statement. If you need to return execution to the main program as the result of a conditional test, use a **subexit** statement. Even when a **subexit** statement is used, the last line of the subprogram must still have a **subend** statement.

Subprograms must always follow the main program and any subroutines used by the main program. Any **image** or **data** statements used by the main program must be located within the main program, and any **image** or **data** statements used by the subprogram must be located within the subprogram.

Information can be passed to a subprogram in two ways: *by reference* or *by value*. Information is passed by reference by putting variable names in the argument list. Passing information by reference allows a subprogram to return information to the main program. Arrays are always passed by reference.

Variable information is passed by value by enclosing it in parentheses in the argument-list of the **call** statement. The value of a constant or calculation is always passed by value.

Compact Computer BASIC provides 17 built-in subprograms that are also executed by the **call** statement.

Review Test 6

1. Which of the following are valid BASIC statements?

   (a) 100 if d="6" then display "DONE!"
   (b) 340 t$=rpt$(15,"*")
   (c) 110 if len(p$)=rpt$(s$,7) then 100
   (d) 400 r=len(seg$(t$,x,y))
   (e) 501 new$=rpt$(old$,3)&seg$(old$,3,2)
   (f) 110 if a<7 and >14 then read a$
   (g) 220 if xor cost>100 then markup=.15
   (h) 300 if not not not a=5 then 9000
   (i) 310 if name$="end" and (b=5 or c=1) then 120
   (j) 660 on n$ goto 10,70,170
   (k) 500 on sgn(x)+2 goto 100,200,300
   (l) 100 on price<100 gosub 200 else 250
   (m) 950 sub price-list(volume,price)
   (n) 700 if x=15 then subend
   (o) 200 call "price"&"list"(x,y,z)

2. What is an ASCII value and why is it important in programming?

3. Which of the following string comparisons are true?

   (a) "Bill Smith" < "Bill Smithe"
   (b) "6" < "5"
   (c) "ASCII" > "ascii"
   (d) "null" <> ""
   (e) "<" < ">"
   (f) "Store1" >= "Store2"
   (g) "dysphemia" > "dysphonia"

(h) "**********" = "*********"

(i) "#!$%&" > "#'$%&"

(j) "phantasmagoria"="phantas"&"magoria"

4. What is the maximum length of a string?

5. The English alphabet is made up of 21 consonants and 5 vowels. Write a program that **reads** the consonants into an array named const$ and the vowels into an array named vowel$. Then have the program construct and display random words consisting of a consonant, a vowel, and a consonant.

6. Write a program that allows you to input a string value and then displays that value in "reverse" order (i.e., last character first, next-to-last character second, etc.). Hint: Use the **seg$** function to take the string apart one character at a time, beginning with the last character in the string, and use the concatenation operation to rebuild it.

7. Write a program that requests the capital city of the countries listed below, and then checks the answer that is entered to see if it is correct.

Countries	**Capitals**
Afghanistan	Kabul
Brazil	Brasilia
Egypt	Cairo
England	London
Ethiopia	Addis Ababa
France	Paris
Greece	Athens
India	New Delhi
Japan	Tokyo
Mexico	Mexico City
United States	Washington
U.S.S.R.	Moscow

8. What does the phrase "levels of precedence" mean when applied to the logical operators?

9. How are subroutines and subprograms similar? How do they differ?

Appendix A

Table of ASCII Codes and Characters

This appendix lists the ASCII codes used by the Compact Computer 40. The decimal and hexadecimal values of the codes are given in the columns titled DEC and HEX. The definitions of the codes are given in the column titled Character (notice that only ASCII codes 32 through 126 have single character definitions). The characters displayed by the **CHR$** function for the displayable codes are shown in the column titled Displayed Using CHR$. The keys used to generate the codes are shown in the column titled Key Sequence.

The user-defined character codes (0–6) and the user-assigned keys (codes 128–137) are shown as two asterisks (**).

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
00	00	NULL	**	CTL 0
01	01	SOH	**	CTL A
02	02	STX	**	CTL B
03	03	ETX	**	CTL C
04	04	EOT	**	CTL D
05	05	ENQ	**	CTL E
06	06	ACK	**	CTL F
07	07	BEL		CTL G
08	08	BS		CTL H
09	09	HT		CTL I
10	0A	LF		CTL J

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
11	0B	VT		$\boxed{\text{CTL}}$ K
12	0C	FF		$\boxed{\text{CTL}}$ L
13	0D	CR		$\boxed{\text{CTL}}$ M or $\boxed{\text{ENTER}}$
14	0E	SO		$\boxed{\text{CTL}}$ N
15	0F	SI		$\boxed{\text{CTL}}$ O
16	10	DLE		$\boxed{\text{CTL}}$ P
17	11	DC1		$\boxed{\text{CTL}}$ Q
18	12	DC2		$\boxed{\text{CTL}}$ R
19	13	DC3		$\boxed{\text{CTL}}$ S
20	14	DC4		$\boxed{\text{CTL}}$ T
21	15	NAK		$\boxed{\text{CTL}}$ U
22	16	SYN		$\boxed{\text{CTL}}$ V
23	17	ETB		$\boxed{\text{CTL}}$ W
24	18	CAN		$\boxed{\text{CTL}}$ X
25	19	EM		$\boxed{\text{CTL}}$ Y
26	1A	SUB		$\boxed{\text{CTL}}$ Z
27	1B	ESC		$\boxed{\text{CTL}}$ $\boxed{\text{CLR}}$
28	1C	FS		$\boxed{\text{CTL}}$ =
29	1D	GS		$\boxed{\text{CTL}}$;
30	1E	RS		$\boxed{\text{CTL}}$.
31	1F	US		$\boxed{\text{CTL}}$,
32	20	Space	Space	Space
33	21	!	!	$\boxed{\text{SHIFT}}$!
34	22	"	"	$\boxed{\text{SHIFT}}$ "
35	23	#	#	$\boxed{\text{SHIFT}}$ #
36	24	$	$	$\boxed{\text{SHIFT}}$ $
37	25	%	%	$\boxed{\text{SHIFT}}$ /
38	26	&	&	$\boxed{\text{SHIFT}}$ &
39	27	'	'	$\boxed{\text{SHIFT}}$ '
40	28	(	(	$\boxed{\text{SHIFT}}$ (
41	29	)	)	$\boxed{\text{SHIFT}}$)
42	2A	*	*	*
43	2B	+	+	+
44	2C	,	,	,
45	2D	–	–	–
46	2E	.	.	.
47	2F	/	/	/

(continued)

(continued)

ASCII Code DEC	HEX	Character	Displayed Using CHR$	Key Sequence
48	30	0	0	0
49	31	1	1	1
50	32	2	2	2
51	33	3	3	3
52	34	4	4	4
53	35	5	5	5
54	36	6	6	6
55	37	7	7	7
56	38	8	8	8
57	39	9	9	9
58	3A	:	:	SHIFT :
59	3B	;	;	;
60	3C	<	<	SHIFT ,
61	3D	=	=	=
62	3E	>	>	SHIFT .
63	3F	?	?	SHIFT ?
64	40	@	@	CTL 2
65	41	A	A	SHIFT A
66	42	B	B	SHIFT B
67	43	C	C	SHIFT C
68	44	D	D	SHIFT D
69	45	E	E	SHIFT E
70	46	F	F	SHIFT F
71	47	G	G	SHIFT G
72	48	H	H	SHIFT H
73	49	I	I	SHIFT I
74	4A	J	J	SHIFT J
75	4B	K	K	SHIFT K
76	4C	L	L	SHIFT L
77	4D	M	M	SHIFT M
78	4E	N	N	SHIFT N
79	4F	O	O	SHIFT O
80	50	P	P	SHIFT P
81	51	Q	Q	SHIFT Q
82	52	R	R	SHIFT R
83	53	S	S	SHIFT S
84	54	T	T	SHIFT T

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
85	55	U	U	$\boxed{\text{SHIFT}}$ U
86	56	V	V	$\boxed{\text{SHIFT}}$ V
87	57	W	W	$\boxed{\text{SHIFT}}$ W
88	58	X	X	$\boxed{\text{SHIFT}}$ X
89	59	Y	Y	$\boxed{\text{SHIFT}}$ Y
90	5A	Z	Z	$\boxed{\text{SHIFT}}$ Z
91	5B	[	[	$\boxed{\text{CTL}}$ 8
92	5C	¥	¥	$\boxed{\text{CTL}}$ /
93	5D	]	]	$\boxed{\text{CTL}}$ 9
94	5E	^	^	$\boxed{\text{SHIFT}}$ ∧
95	5F	_	_	$\boxed{\text{CTL}}$ 5
96	60	`	`	$\boxed{\text{CTL}}$ 3
97	61	a	a	A
98	62	b	b	B
99	63	c	c	C
100	64	d	d	D
101	65	e	e	E
102	66	f	f	F
103	67	g	g	G
104	68	h	h	H
105	69	i	i	I
106	6A	j	j	J
107	6B	k	k	K
108	6C	l	l	L
109	6D	m	m	M
110	6E	n	n	N
111	6F	o	o	O
112	70	p	p	P
113	71	q	q	Q
114	72	r	r	R
115	73	s	s	S
116	74	t	t	T
117	75	u	u	U
118	76	v	v	V
119	77	w	w	W
120	78	x	x	X
121	79	y	y	Y

(continued)

(*continued*)

ASCII Code			Displayed
DEC	HEX	Character	Using CH
122	7A	z	z
123	7B	{	{
124	7C	|	|
125	7D	}	}
126	7E	→	→
127	7F	DEL	÷
128	80	* *	
129	81	* *	
130	82	* *	
131	83	* *	
132	84	* *	
133	85	* *	
134	86	* *	
135	87	* *	
136	88	* *	
137	89	* *	
138	8A		
139	8B		
140	8C		
141	8D		
142	8E		
143	8F		
144	90		
145	91		
146	92		
147	93		
148	94	DELETE	
149	95		
150	96		
151	97	NUMBER	
152	98	VERIFY	
153	99	SAVE	
154	9A	OLD	
155	9B	LIST	
156	9C	CALL	
157	9D	ELSE	
158	9E	CHR$(	

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
159	9F	GOTO		FN =
160	A0			FN CLR
161	A1	ASN(	｡	FN A
162	A2	PAUSE	｢	FN B
163	A3	GRAD	｣	FN C
164	A4	ATN(	､	FN D
165	A5	TAN(	･	FN E
166	A6	LN(	ｦ	FN F
167	A7	LOG(	ｧ	FN G
168	A8	LINPUT	ｨ	FN H
169	A9	NEXT	ｩ	FN I
170	AA	INPUT	ｪ	FN J
171	AB	PRINT	ｫ	FN K
172	AC	USING	ｬ	FN L
173	AD	THEN	ｭ	FN M
174	AE	IF	ｮ	FN N
175	AF	GOSUB	ｯ	FN O
176	B0	RETURN	ｰ	FN P
177	B1	SIN(	ｱ	FN Q
178	B2	PI	ｲ	FN R
179	B3	ACS(	ｳ	FN S
180	B4	SQR(	ｴ	FN T
181	B5	TO	ｵ	FN U
182	B6	EXP(	ｶ	FN V
183	B7	COS(	ｷ	FN W
184	B8	RAD	ｸ	FN X
185	B9	FOR	ｹ	FN Y
186	BA	DEG	ｺ	FN Z
187	BB	BREAK	ｻ	FN BREAK
188	BC		ｼ	SHIFT RUN
189	BD		ｽ	CTL RUN
190	BE	CONTINUE	ｾ	FN RUN
191	BF	RUN	ｿ	RUN
192	C0		ﾀ	SHIFT FN 0
193	C1		ﾁ	SHIFT FN 1
194	C2		ﾂ	SHIFT FN 2
195	C3		ﾃ	SHIFT FN 3

(continued)

(continued)

ASCII Code			Displayed	Key
DEC	HEX	Character	Using CHR$	Sequence
196	C4		ﾄ	SHIFT FN 4
197	C5		ﾅ	SHIFT FN 5
198	C6		ﾆ	SHIFT FN 6
199	C7		ﾇ	SHIFT FN 7
200	C8		ﾈ	SHIFT FN 8
201	C9		ﾉ	SHIFT FN 9
202	CA		ﾊ	
203	CB		ﾋ	
204	CC		ﾌ	
205	CD		ﾍ	
206	CE		ﾎ	
207	CF		ﾏ	
208	D0		ﾐ	
209	D1		ﾑ	
210	D2		ﾒ	
211	D3		ﾓ	
212	D4		ﾔ	
213	D5		ﾕ	
214	D6		ﾖ	
215	D7		ﾗ	
216	D8		ﾘ	
217	D9		ﾙ	
218	DA		ﾚ	
219	DB		ﾛ	
220	DC		ﾜ	
221	DD		ﾝ	
222	DE		ﾞ	
223	DF		ﾟ	
224	E0		α	
225	E1		ä	
226	E2		β	
227	E3		ε	
228	E4		μ	
229	E5	PB	σ	SHIFT ▲
230	E6	OFF	ρ	OFF
231	E7	BREAK	g	BREAK
232	E8	UP	√	▲

(continued)

(*continued*)

ASCII Code			Displayed	Key
DEC	**HEX**	**Character**	**Using CHR$**	**Sequence**
233	E9	DOWN	⁻i	▼
234	EA	SHIFT	⁞	SHIFT ENTER
235	EB		⁘	
236	EC		φ	
237	ED		£	
238	EE		ñ	
239	EF		ö	
240	F0		α	
241	F1		a	
242	F2		β	
243	F3		≈	
244	F4		ä	
245	F5		ü	
246	F6	DEL	Σ	SHIFT ◄
247	F7	INS	π	SHIFT ►
248	F8	HOME	x̄	CTL ▲
249	F9	SKIP	μ	CTL ▼
250	FA	CLR	‡	CLR
251	FB	BTAB	╦	CTL ◄
252	FC	←	╤	◄
253	FD	FTAB	÷	CTL ►
254	FE	→		►
255	FF			

Appendix B

List of Compact Computer BASIC Reserved Words

This appendix provides an alphabetic listing of the CC-40 BASIC reserved words. A reserved word cannot be used as a valid variable name, but can be part of a variable name.

ABS
ACCEPT
ACS
ALL
ALPHA
ALPHANUM
AND
APPEND
ASC
ASN
AT
ATN
ATTACH
BEEP
BREAK
CALL
CHR$
CLOSE
CON (abbreviation for CONTINUE)
CONTINUE
COS
DATA
DEG
DEL (abbreviation for DELETE)
DELETE
DIGIT
DIM
DISPLAY
ELSE
END
EOF
ERASE
ERROR
EXP
FOR
FORMAT
FRE
GOSUB
GOTO
GRAD
IF
IMAGE
INPUT
INT
INTERNAL
INTRND
KEY$
LEN
LET
LINPUT
LIST
LN
LOG
NEW
NEXT
NOT
NULL
NUM (abbreviation for NUMBER)
NUMBER
NUMERIC

OLD
ON
OPEN
OR
OUTPUT
PAUSE
PI
POS
PRINT
PROTECTED
RAD
RANDOMIZE
READ
REC
RELATIVE
RELEASE
REM
REN (abbreviation
for RENUMBER)
RENUMBER
RESTORE
RETURN
RND
RPT$
RUN
SAVE
SEG$
SGN
SIN
SIZE
SQR
STEP
STOP
STR$
SUB
SUBEND
SUBEXIT
TAB
TAN
THEN
TO
UALPHA
UALPHANUM
UNBREAK
UPDATE
USING
VAL
VALIDATE
VARIABLE
VERIFY
WARNING
XOR

Appendix C

Answers to Review Questions

REVIEW TEST 1

1. (a) Valid
   (b) Valid
   (c) Invalid (period not allowed in a variable name)
   (d) Valid
   (e) Valid
   (f) Invalid (# sign not allowed in a variable name)
   (g) Valid
   (h) Invalid (variable names cannot begin with a digit)
   (i) Valid
   (j) Invalid (hyphen not allowed in a variable name)
   (k) Invalid (**end** is a reserved word)
   (l) Valid
   (m) Valid
   (n) Invalid (variable names cannot begin with a digit)

2. Variable names must begin with a letter of the alphabet, underline character, or @ sign; cannot be more than 15 characters long; and cannot be on the reserved word list. String variables must end with a $ sign.

3. (a) Valid ("" is the null string)
   (b) Valid

(c) Invalid (A space must separate 100 from meters if 100 is intended as a line number. Otherwise, 100meters is an invalid variable.)
(d) Invalid (cannot assign values to a constant)
(e) Valid (the **let** is optional)
(f) Valid
(g) Valid
(h) Invalid (the variable must be on the left side of the = sign)
(i) Valid
(j) Valid (**old** and **new** are on the reserved word list—old$ and new$ are not)

4. The initial value of a numeric variable is zero; the initial value of a string variable is null.

5. The display buffer is 80 characters wide. Although only 31 characters can be seen at one time, the entire buffer contents can be viewed by scrolling the display window right and left with the $\boxed{\blacktriangleright}$ and $\boxed{\blacktriangleleft}$ keys.

6. The **display** 1,,3 statement is a legal BASIC sequence that places 1 in the first print zone and 3 in the third print zone. The second print zone is left blank.

7. The comma print-list separator instructs the computer to display the next print item in the next print zone. The semicolon instructs the computer to display the next item immediately following the preceding item.

8. (a) Invalid (0 is not a valid column position)
   (b) Valid
   (c) Valid
   (d) Valid (Using seven commas instructs the computer to display the 7 in the seventh display zone. Since there are only six display zones in the display buffer, the seventh

zone is actually the first zone of the next display cycle. You must press $\boxed{\text{ENTER}}$ to see the next cycle.)

(e) Invalid (only one **at** parameter allowed per **display** statement)

(f) Valid

(g) Invalid (**display** is not a valid variable name)

9. No, zero is not a valid line number.

10. To insert a new line, assign it a line number between the existing line numbers where you want it to appear. If a program is numbered with no gaps between the lines, you cannot insert a new line unless you renumber the program. The **renumber** command is discussed in Lesson 9.

11. To replace a line, enter the new line with the same line number as the old.

12. Use the **new** command to erase an entire program from memory. To erase a line, enter **delete** (or **del**) and the line number.

13. The writer evidently intended to display "Humphrey Bogart". Without a **pause** statement, the name will not be displayed long enough to be seen.

REVIEW TEST 2

1. (a) Invalid (the prompt is not enclosed in quotes)

   (b) Invalid (a comma is used to separate multiple **input** variables—not a semicolon)

   (c) Invalid (This example mixes two types of assignment statements—the **input** statement which assigns keyboard input to a variable, and the assignment sequence which assigns a value to a variable.)

   (d) Valid (the string-expression ID$&"−001" is a valid prompt)

(e) Valid
(f) Invalid (a semicolon must separate a prompt and an **input** variable)

2. A **pause** 0 statement cancels a **pause all** statement.

3. A program that counts by 9s in column 9 is shown below.

```
100 DISPLAY AT(9),X
110 PAUSE .2
120 X=X+9
130 GOTO 100
```

   A program that counts backwards from 2475 by 99 and beeps each time a number is displayed is shown below.

```
100 X=2475
110 DISPLAY AT(9) BEEP,X
120 PAUSE .2
130 X=X-99
140 GOTO 110
```

4. A *pending display state* instructs the computer to preserve the present contents of the display when the next **display** statement is executed. It is created by placing a semicolon or comma after a **display** statement. The advantage of the pending display state is that it allows information to be jointly displayed by different **display** statements.

5. The completed program is given below.

```
100 INPUT "Current balance? ";BALANCE
110 INPUT "Amount of check or deposit? ";AMOUNT
120 BALANCE=BALANCE+AMOUNT
130 DISPLAY "New balance = $";BALANCE
140 PAUSE
150 GOTO 110
```

A sample **run** of the program is shown below.

```
Enter      Display
run        Current balance?
127.18     Amount of check or deposit?
-29.95     New balance = $ 97.23
[ENTER]    Amount of check or deposit?
-4.51      New balance = $ 92.72
[ENTER]    Amount of check or deposit?
-67.12     New balance = $ 25.6
[ENTER]    Amount of check or deposit?
-49.95     New balance = $-24.35
[ENTER]    Amount of check or deposit?
300        New balance = $ 275.65
[BREAK]    Break
```

6. A "counting variable" is commonly used to control the number of times a loop is executed. The counting variable is set to some initial value before the loop is entered and is increased or decreased by one each time the loop is executed. An **if then** test is used to determine when a specified termination value has been reached. When the termination value is reached, execution is transferred out of the loop.

7. (a) Valid (While valid, notice that the test x=x+1 can never have a true result. An **if then** tests the truth or falsehood of a condition; it does not perform an assignment operation.)
   (b) Valid (true and false are valid variable names)
   (c) Valid (The not-equal-to sign can be entered as <> or ><. The former is the customary way to enter it.)
   (d) Invalid (this type of relational expression is not valid in BASIC)
   (e) Valid
   (f) Valid
   (g) Valid (an **if then** statement is valid as the action of another **if then** statement)

8. A program that solves this problem is shown below.

```
100 INPUT "Enter order: ";ORDER
110 COST=ORDER*14.88
120 IF ORDER=>5000 THEN DISCOUNT=COST*.4
130 IF ORDER<5000 THEN DISCOUNT=COST*.35
140 IF ORDER<=1000 THEN DISCOUNT=COST*.3
150 IF ORDER<=500 THEN DISCOUNT=COST*.25
160 IF ORDER<=100 THEN DISCOUNT=COST*.2
170 PRICE=COST-DISCOUNT
180 DISPLAY "Wholesale price = $";PRICE
190 PAUSE
200 GOTO 100
```

A sample **run** appears below.

```
Enter      Display
run        Enter order:
433        Wholesale price = $ 4832.28
ENTER      Enter order:
888        Wholesale price = $ 9249.408
ENTER      Enter order:
2001       Wholesale price = $19353.672
BREAK      Break
```

REVIEW TEST 3

1. The phrase "levels of precedence" refers to the ranking system used by the computer to determine the order in which mathematical operations are performed. The function of parentheses in a calculation is to force the computer to evaluate an operation or series of operations in an order different from that resulting from the levels of precedence.

2. The square root of a number can be calculated by raising the value to the ½ power. The fifth root can be calculated by raising the value to the ⅕ power.

3. The unary minus operation will be performed first.

(cont. from preceding page)

```
220 total=total+grade(outerloop,innerloop)
230 next innerloop
240 average=total/3
250 display "Student";outerloop;"average =";average
260 pause
270 next outerloop
280 !test scores
290 data 78,82,89
300 data 83,85,81
310 data 100,70,73
320 data 65,68,71
330 data 99,89,93
340 data 87,0,93
```

A sample **run** is shown below.

```
Enter     Display
run       Student 1 average = 83
ENTER     Student 2 average = 83
ENTER     Student 3 average = 81
ENTER     Student 4 average = 68
ENTER     Student 5 average = 93.66666667
ENTER     Student 6 average = 60
```

REVIEW TEST 6

1. (a) Invalid (cannot compare numeric and string values)
   (b) Invalid (the **rpt$** argument parameters are reversed)
   (c) Invalid (cannot compare numeric and string values)
   (d) Valid
   (e) Valid
   (f) Invalid (the second comparison is not complete—it should be a>14)
   (g) Invalid (the **xor** operator requires two tests, not one)
   (h) Valid (one **not** can perform the identical test: **not** a=5)
   (i) Valid
   (j) Invalid (the **on** statement requires a numeric decision value)
   (k) Valid

(l) Invalid (the line is a mixture of **if then** and **on** statements)
(m) Invalid (subprogram names cannot contain hyphens)
(n) Invalid (The **subend** statement cannot appear in an **if then** statement. The **subexit** statement must be used in this type of construction.)
(o) Invalid (subprogram names must be constants, not string-expressions)

2. String characters are represented inside the computer as numbers between 0 and 255. These numbers are referred to as ASCII values (ASCII is an acronym for American Standard Code for Information Interchange). When the computer compares strings for decision-making purposes, it compares the ASCII values of the characters in the strings.

3.

(a)	true	(f)	false
(b)	false	(g)	false
(c)	false	(h)	false
(d)	true	(i)	false
(e)	true	(j)	true

4. The maximum length of a string is 255 characters.

5. A program to display randomly constructed three-letter words is shown below. (Programs like this are sometimes used by companies to generate a list of potential names for new products.)

```
100 !random 3-letter words
110 randomize
120 dim const$(21),vowel$(5)
130 for count=1 to 21
140 read const$(count)
150 next count
160 for count=1 to 5
170 read vowel$(count)
180 next count
```

(*cont. on following page*)

(*cont. from preceding page*)

```
190 x=intrnd(21)
200 y=intrnd(5)
210 z=intrnd(21)
220 display const$(x);vowel$(y);const$(z)
230 pause .5
240 goto 190!loop
250 data, b,c,d,f,g,h,j,k,l,m,n,p,q,r,s,t,v,w,x,y,z
260 data a,e,i,o,u
```

A sample **run** is shown below.

```
Enter      Display
run        yeg
           duc
           qak
           wer
           xax
[BREAK]    Break
```

6. A program to reverse the order of characters in a string is shown below.

```
100 !reverse string order program
110 input "Enter sample string: ";word$
120 reverse$=""
130 lettercount=len(word$)
140 for loop=lettercount to 1 step -1
150 reverse$=reverse$&seg$(word$,loop,1)
160 next loop
170 display reverse$
180 pause
190 goto 110
```

A sample **run** is shown below.

```
Enter        Display
run          Enter sample string:
qwerty       ytrewq
[ENTER]      Enter sample string:
MXYZTPLK     KLPTZYXM
```

```
ENTER            Enter sample string:
1234567890       0987654321
BREAK            Break
```

7. A program that tests for the capitals is shown below.

```
100 !capitals of the world
110 pause all
120 for count=1 to 13
130 read country$,capital$
140 input "Capital of "&country$&"? ";guess$
150 if guess$<>capital$ then display "Not correct. Try
    again!":goto 140
160 display guess$;" is correct!"
170 next count
180 end
190 data Afghanistan,Kabul,Brazil,Brasilia,Egypt,Cairo
200 data England,London,Ethiopia,Addis Ababa,France,
    Paris
210 data Greece,Athens,India,New Delhi,Japan,Tokyo
220 data Mexico,Mexico City,United States,Washington
230 data U.S.S.R.,Moscow
```

8. The phrase "levels of precedence" refers to the order in which operations are performed. For the logical operators, this order is: **not**, **xor**, **and**, and **or** (from highest to lowest priority).

9. Subroutines and subprograms are similar in that they can be executed without permanently changing the flow of program execution. When the subroutine or subprogram is finished, execution resumes with the first program instruction following the statement that executed the subroutine or subprogram. Subroutines and subprograms differ in their use of variables. Subroutines use the same variables as the main program, while subprograms use an entirely separate set of variables. Therefore, subprograms require that information be passed between the main program and a subprogram by means of optional parameters.

Index

Computers/**VTX Series**

LEARN BASIC: A Guide to Programming the Texas Instruments Compact Computer 40

DAVID THOMAS

This book is all you'll need to program your Texas Instruments hand-held computer in the simple-to-learn BASIC programming language. LEARN BASIC is organized into 28 short, easy-to-follow lessons. Each lesson covers an important programming concept or a BASIC statement or command. Review exercises, designed to reinforce what you've learned, follow every 4 or 5 lessons.

- **Designed for people who want to learn BASIC fast and have no previous computer experience.**
- **Uses a practical, hands-on approach to this most popular of computer programming languages.**
- **Contains many example programs that can be used by engineering and scientific professionals, students, business people, managers, hobbyists, and computer enthusiasts.**
- **Includes complete answers to all exercises.**
- **Allows you to begin learning immediately, as it contains no long technical or theoretical discussions.**
- **Covers all the major statements and commands of Compact Computer BASIC.**

As a lesson-oriented book, LEARN BASIC provides heavy emphasis on examples, work problems, review exercises, and user interaction that allows you to *learn by doing*.

David Thomas is a writing consultant to Texas Instruments with many years of experience in writing for programmable products and computers.

TEXAS INSTRUMENTS
INCORPORATED

ISBN 0-07-064257-5