Apple II

A Touch of Applesoft BASIC

Apple IIc, Apple IIe, Apple IIGS™

Apple® II A Touch of Applesoft BASIC

Printed in Singapore.

Contents

Preface

This tutorial will help you get started writing simple Applesoft BASIC computer programs on your Apple® II computer. You won't learn all there is to know about Applesoft BASIC from just this tutorial; but by the time you finish these ten sessions, you'll be able to decide whether you want to continue learning about programming.

The product training disk that came with your computer gives you a brief introduction to Applesoft; you might want to work with that disk before you read this tutorial.

What's a computer language?

A computer language is like the languages that people speak. It has a vocabulary and a syntax—word order is important and spelling counts. Your Apple computer speaks a language called **Applesoft BASIC.** (It speaks other languages, too, but they aren't built into the computer; you buy them on disks.) The computer reads the BASIC instructions you type from the keyboard, and then it does exactly what it's told. Luckily, it's easier to learn BASIC than a human language because BASIC has far fewer words, and its grammar is usually very straightforward.

❖ *BASIC by any other name ...* There are many variations on the BASIC computer language. But in this little tutorial the terms *BASIC, Applesoft BASIC,* and *Applesoft* all refer to the same thing.

What's a program?

Computer programming is writing instructions for your computer. The entire set of instructions you give to a computer to make it do something is the **program**. Imagine that your computer is a pet you want to train. You can't talk to your pet in the same way you talk with a human; you have to use a limited vocabulary to tell it exactly what to do. If you wanted it to do a series of things, you would give it a set of instructions, one instruction at a time. For instance, suppose you want your pet to sit, lie down, and roll over. You'd do it like this:

"King, sit."

(King sits)

"King, lie down."

(King lies down)

"King, roll over."

(King rolls over)

"Good dog!"

(King wags tail)

Of course your Apple won't sit, lie down, or roll over, but it will do a lot of things for you if you give it instructions in a systematic and logical order. You use the same kind of directness, simplicity, and order in computer programming as in pet training (except that you don't have to praise your computer when it does what you tell it).

Do you have to program?

You don't have to write programs to use your computer. Thousands of programs have already been written for your Apple—programs for word processing, financial analysis, computerized file cabinets, and dozens of other applications. You just put a disk with programs on it into your disk drive and turn on your computer.

Why would you want to learn to program?

First of all, you might find programming to be a lot of fun. When you learn to program, you discover that your Apple isn't really magical (although it certainly seems that way at times); it's just following the instructions that you give it. When you program your computer, you make it do what *you* want it to do—you get to create your own magic. Second, you learn a lot about how a computer works as you learn to program it. That gives you a better understanding of what your computer can and can't do. Finally, you might find that programming is something that really intrigues you and stimulates your own creativity in ways you'd never thought about. You might eventually decide to become a professional programmer.

You can create simple entertainment, educational, and business programs with just an elementary set of instructions. For example, you can write very effective educational games in Applesoft BASIC, or even home budgeting and checkbook programs to keep your finances in order.

Writing your own program is an *option* available on your Apple. While you're likely to find programming useful and interesting, you don't have to learn how to program to use your computer. But if you *do* want to program, you'll find Applesoft BASIC a great place to start.

Patience required

Learning to program is a little like learning how to become a chef. You've got to be an experienced chef to pull off great seven-course meals; but the essentials of the craft begin with melting butter, turning an egg, and so on. And the payoff is similar, too. You don't have to be a master chef to enjoy a homemade omelette (or amaze your friends with your culinary prowess).

From time to time, you'll just have to be patient—but only for a little while. Have faith.

How to get started

Applesoft is built into your Apple II computer. But you need to prepare your computer to store the programs you create so that you can use them again. (You'll learn more about storing your programs onto disk in Session 4.) Here are the steps to take to begin your study of Applesoft BASIC:

1. Read your Apple computer owner's guide first. It contains lots of valuable information about the computer that you'll need to know before you can begin to use Applesoft. Pay special attention to the section on formatting disks. You'll need at least one formatted disk before you can start.
2. Insert the utilities disk that came with your computer into the disk drive, close the disk drive door, and turn on the computer. (See your owner's manual for instructions.) Choose the Applesoft BASIC option and press Return; you should see this symbol:]
3. Remove the disk from the drive and replace it with a formatted disk. Be sure to close the drive door.

❖ *Using Applesoft without a disk drive:* If you don't have a disk drive, you can still write programs; but you won't be able to store them. To start BASIC without a disk drive, turn on your computer and then press the Control and Reset keys at the same time, then release them. You'll see this symbol:].

And now—begin!

This tutorial is divided into ten sessions; you'll need about an hour for each session. Be sure to spend lots of time practicing what you've learned in each session before going on to the next one; each session builds on the previous one.

Above all, have a good time. Experiment as much as you can. Break the rules. Try crazy things—the worst thing that can happen is that the computer will beep at you. (When this happens, beep back.)

Now, all you have to do is turn the page and begin.

Session 1

Getting Started

The best way to find out if you like programming is to do some. To keep things simple, do everything exactly as it's presented in this tutorial. Of course if you get bored, strike out on your own! You won't break the computer by typing something wrong, and the important thing is to experiment, learn, and have fun.

In this first session, you'll learn the rudiments. You'll read about program lines and line numbers, and how to type in programs. You'll see how to put messages on the screen with the PRINT instruction, and you'll learn some things about programming mistakes and how to fix them.

The elementary stuff

Before you do anything else, type the word NEW and press the Return key. **NEW** tells your Apple computer to make way for a new program. Pressing Return tells your Apple to look at what you just typed. Until you press Return, your Apple thinks you're just talking to yourself:

NEW ———— Press Return here.

Now type the following line exactly as you see it, and then press Return:

10 PRINT "SIT" ———— Press Return here.

The number 10 is called a **line number**. Your Apple executes the lines of instructions you type in numeric order, always beginning with the lowest number. For the time being, number your program lines by 10's. You'll learn why later in Session 3.

After you've typed all the instructions (which you've just done—your first program is a short one), type RUN and press Return. The **RUN** command tells your Apple that you've finished giving it instructions and that you want it to carry them out:

RUN ———— Press Return here.

Your video display should look something like this:

```
]NEW
]10 PRINT "SIT"
]RUN
SIT
]▌
```

You've just written and **executed** (another word for *run*) your first computer program. Congratulations! You've also just learned one of the most often used programming instructions: PRINT. The PRINT instruction tells your computer to display whatever appears within quotation marks. Here's some more practice using PRINT. Type the following program exactly as it appears. (If you make a mistake, just press Return and retype the line.) Be sure to press Return at the end of each line:

```
10 print "lie down"
20 Print "Roll Over"
30 pRiNt "GeT wEiRd"
RUN
```

You'll see this on your screen:

```
lie down
Roll  Over
GeT wEiRd
```

❖ *Why you don't need NEW here:* When you re-use a line number, the new line replaces the old one. The last program you typed had only one line—line 10. This new program also has a line 10, replacing the old one. It's as if you'd typed NEW anyway.

Your computer doesn't care whether the letters are uppercase or lowercase, or some combination of both. But you've got to be careful how you type your instructions. Your computer expects to be told exactly what to do *in a way that it can understand* or you'll get an error message like this one:

```
?SYNTAX ERROR IN 10
```

Computers always do *exactly* what you say, not necessarily what you *mean* to say. Even minor typing errors will bring up a syntax error message (usually with a line number to help you find the error). Type:

```
NEW
```

and press Return; then type this one-line program and try running it:

```
10 PRIMT "WHOOPS" ———————— Watch out!!
```

(Be sure to press Return at the end of the line—this is your last reminder.)

After you run the program, you'll see this message:

```
?SYNTAX ERROR IN 10 ———————— 10 is the line number.
```

Even though you and any other human who saw it would know that you *meant* PRINT instead of PRIMT, the instruction baffled your Apple. Luckily, most mistakes make your computer show a built-in error message that will tell you what you did wrong. As you program more (and, naturally, make more mistakes along the way), you'll see more messages to help you understand how your computer operates. *Remember:* the computer displays error messages to help you correct mistakes, not to tell you you're a dummy. Treat these messages as helpful guides and not as nagging annoyances.

Editing: program first aid

You've just seen that you have to be careful when you enter a computer program to avoid introducing a **bug**, or error. Many bugs are the result of simple typing errors; you can avoid a lot of debugging later by checking your typing as you go along.

Retyping a whole line every time you make a simple typing error gets tiresome very quickly. Your Apple has some built-in features to make debugging easier.

Type the following line, but don't press Return yet:

```
10 PRINT K "LOOK OUT, YOU BUG"
```

Don't press Return yet!

That *K* between the PRINT instruction and the message is going to cause problems. You *could* re-type the whole line, but if you had to do that every time you made an error, you'd never get anything done. Instead, locate the four arrow keys in the lower-right corner of your keyboard. Then do this:

1. Press the Left-Arrow key until the cursor is directly over the offending K.
2. Press the Space bar once to erase the K (don't use the Delete key; it won't work with Applesoft).
3. Using the Right-Arrow key, move the cursor until it is to the right of the last quotation mark in the line. (If you press Return in the middle of the line, you'll lose everything from that point to the line's end.)
4. Now check and make sure your line is correct.

Your line should look like this:

```
10 PRINT   "LOOK OUT, YOU BUG"
```

Now you can press Return and run the program; it'll work fine.

❖ *The origins of bug* Back in the old days, computers used vacuum tubes, had a million miles of wires, and required large, air-conditioned rooms to keep them working. Computer folklore has it that one day a moth got into the computer room and flew into the computer. The moth was fried to a crisp, but it didn't die alone—its demise brought the computer to a dead stop. After searching high and low to find what caused the computer to "crash," a programmer found the moth's remains and announced (with no regard for genus or phylum), "Hey. There's a bug in the computer." The rest is history.

Summary and review

In this first session, you learned how to make way for new programs with NEW, how to execute programs with RUN, and how to put messages on the screen with PRINT. You saw how programs use line numbers to arrange the sequence of instructions. Finally, you learned a few things about bugs and how to get rid of them.

Before you go on to the next session, experiment with the PRINT instruction. Write a five-line program; then change the line numbers by retyping the lines (making the last line the first one, for example) to see what happens. And don't be afraid to make mistakes—nobody's keeping score!

Session 2

Arithmetic and Variables

You don't have to know a lot about arithmetic to learn to program your Apple computer. But most programs require arithmetic functions to make them work. (For example, in a checkbook balancing program you might want to subtract the amount of each check that you write from the account balance.) In this session, you'll learn the basics of computer arithmetic. You'll also read about variables, the storage areas in the computer's memory that hold values. Finally, you'll learn the rules for giving names to variables to make them easier to handle—and then you'll be encouraged to break the rules to see what happens.

Arithmetic

You learned in the first session that your Apple displays anything enclosed in quotation marks after the PRINT instruction. To do arithmetic, use the PRINT instruction *without* quotation marks.

For example, type this program and run it:

```
NEW
10 PRINT "5 + 5"
20 PRINT 5 + 5
RUN
5 + 5      Line 10 printed exactly what was inside the quotation marks.
10         Line 20 printed the sum of the two numbers.
```

In the first line, you told your Apple to print the phrase 5 + 5. But in the second line, you said, "Add the numbers 5 plus 5, and show the answer on the screen."

As you might expect, your Apple can do more than just add. In fact, it can do some extremely complex math. But in this tutorial, you'll stick to the basics: addition, subtraction, multiplication, and division. Here's a chart that shows the symbols (called **operators**) your computer uses to do simple arithmetic:

Operator	Action
+	add
–	subtract
*	multiply
/	divide

The addition and subtraction operators are the same ones you've always used. You've probably seen the division operator before, used to express a fraction (as in 7/8). The only one that looks a little different is the multiplication operator; it's an asterisk (*) instead of an *X*. Many programmers use the letter *X* to represent some unknown value, so somebody decided to use the asterisk (which is like an *X* with a horizontal line through its center) instead.

Here's a sample program. Type it; but before you run it, predict what the answers will be:

```
10 PRINT 4 + 5.1
20 PRINT 7.56 - 4.44
30 PRINT 4 * 5
40 PRINT 4.6 / 2
50 PRINT 11 + 12 - 13 + 14
60 PRINT 12 / 3 + 4
70 PRINT 10 * 2 + 8 / 2
```

Here's simple addition.

Your computer handles decimals easily.

Remember: * means multiply.

Here's simple division.

It can do multiple operations.

The computer solves problems from left to right ...

... but there are other considerations (read about precedence in the next section).

Line 20 shows you that your computer can handle fractions—you just need to express them in a way your computer can understand. For example, if you mean to tell your computer to determine the sum of two and one-half plus three by typing this:

```
PRINT 2 1/2 + 3
```

you'll get an answer you hadn't counted on. Your computer will display 13.5 instead of 5.5. It interprets 2 1/2 + 3 as "divide the number 21 by 2; take that answer and add 3 to it." Spaces between numbers mean nothing to your electronic friend.

If you worked out all of the problems in your head before you ran the program, the last answer may have been a surprise:

```
70 PRINT 10 * 2 + 8 / 2
```

The answer is 24, not 14!

The result of the calculations is based on **precedence**. Precedence is the order in which your computer does mathematical operations.

Precedence: the order of calculations

In general, your Apple does calculations from left to right. But all multiplication and division happens before addition and subtraction. Step through the calculations in line 70 to see how precedence works.

Calculation:	10 * 2 + 8 / 2
Step 1:	10 * 2 = 20
Step 2:	8 / 2 = 4
Step 3:	20 + 4 = 24

Use parentheses to change precedence

Sometimes you'll need to re-order precedence so that you can first do addition and subtraction and then do multiplication and division. For example, what if you meant

```
PRINT 18 + 4 / 2
```

to mean you wanted to add 18 and 4 first, and then divide the sum by 2? Look at the following little program to see how to do it:

```
NEW
10 PRINT 18 + 4 / 2 ---------------- This comes out 20...
20 PRINT (18 + 4) / 2 -------------- ...but this comes out 11.
```

Line 10 first handles the division, then adds the result to 18. Line 20 *re-orders* precedence by enclosing the sum within parentheses. Parentheses change the order of precedence. Whatever you type within parentheses is solved first, again from left to right and multiplication/division before addition/subtraction.

If you need to, you can embed parentheses within other parentheses to show precedence in more complex situations. Just remember to go from the innermost set of parentheses and move outward.

Take a look at this next program and see if you can guess what the results will be before you run it:

```
10 PRINT (7-3) * 2
20 PRINT  3 * ((10 - 6) / 2)
30 PRINT ((4 - 3) / (9 + 2)) * 2
40 PRINT (((1 + 2) * (2-1)) + 11) / 10
```

Now run the program and see if you were right.

Whenever you start using a lot of parentheses, check to make sure that the number of left parentheses matches the number of right parentheses. If the totals of left and right parentheses are different, you'll get a syntax error message.

❖ *Pretend you're the computer:* Every time you write a program or a section of a program, run it in your head before you run it in your computer. The more you "play computer," the more you'll understand how your computer operates. As that happens, you'll automatically type instructions the way the computer needs to see them; you'll soon find that you get far fewer error messages. Try it for a while and see what happens.

Experiment with your own arithmetic programs. Try mixing the precedence up. Mix in some phrases to label what you're doing. For example:

```
NEW
10 PRINT "The sum of 12 plus 20, divided by the difference between 5 and 3.5, is "
20 PRINT (12 + 20) / (5 - 3.5)
```

❖ *About unsightly "runover" lines:* If your computer is set to display 40 columns on your screen, line 10's quotation ran over the edge of the screen and wrapped to the next line. The word *divided* was split in the process. As you go along you'll pick up little tricks to avoid such unsightly split words; for the time being, try to ignore them—your computer does.

So now you know how to use your Apple to do arithmetic. And you can use it as you would a calculator (although using a calculator is probably quicker and easier). But the simple arithmetic functions you just learned become much more powerful when you use them with variables.

Variables

Variables are symbols for values. They're called *variables* because their values can change or *vary*. Variables look like phrases you forgot to put in quotation marks:

```
NEW
10 PRINT "HELLO"
20 PRINT HELLO
RUN
HELLO        Line 10 prints this
0            Line 20's work.
```

In this program, the first HELLO is a phrase for the computer to print just as it is. The second HELLO is a variable whose value happens to be zero. You give a value to a variable by using the equal sign(=).

Add these lines to the HELLO program and run it:

```
30 HELLO = 128
40 PRINT HELLO        This will show up as 128!
RUN
HELLO
0
128                   New value for variable HELLO assigned in line 30.
```

You've just assigned the value 128 to a variable called HELLO. Think of a variable as a temporary storage box. Whatever you put into the box stays there until you replace it with something else. Add these two lines to your program and run it again:

```
50 HELLO = 3500
60 PRINT HELLO
RUN
```

You can do math with variables. Try the following program:

```
NEW
10 A = 15
20 B = 95
30 PRINT A + B
```

Variables can hold the result of calculations on other variables as well as on numbers. Type the following program and see if you can guess the results before you run it:

```
10 LOW = 5
20 HIGH = 9
30 SUM= LOW + HIGH
40 PRINT SUM
```

The sum of variables LOW and HIGH ends up in the third variable, SUM.

Try out the following program to see the various combinations of numbers and variables you can get.

```
10 W = 14.5
20 X = 6.5
30 PRINT (W + X) * 2
40 Y = W - X + 3
50 PRINT Y
60 Z = 3 * Y - 2
70 PRINT Z
```

Naming variables

Applesoft imposes a few restrictions on naming variables. Here's a list:

- A variable name must begin with a letter.
- Characters after the first one can be a mixture of letters and digits (no symbols).
- Certain letter combinations (called reserved words) have special meaning to Applesoft and can't be used in any part of a variable name. (You'll learn more about this rule in Session 3.)
- A name can be up to 238 characters long, but the computer recognizes only the first two. (The others are to remind you what the variable stands for.)

When you write a very short and simple program, using single letter variables is a safe way to make sure a variable name doesn't conflict with another variable. (Your computer sees SUM and SUNDAY as the same variable because of the last rule in the chart.) But when you begin writing longer programs, it really helps to have variable names that describe what's going on.

For example, if you're calculating the area of a circle, you'll need the value of pi (π) in your program. You could have the variable X hold the value of pi (3.141592). It makes more sense, though, to give variables more meaningful names:

```
NEW
10 PI = 3.141592
20 RADIUS = 5
30 AREA = PI * RADIUS * RADIUS
40 PRINT AREA
```

Math: $A = \pi R^2$

Descriptive variable names make it easy for you to see what the program is doing when you read your code (a synonym for program).

◆ *Store only numbers in numeric variables.* The kinds of variables you're learning about now are called **numeric variables.** That means that you can use them only to hold the value of numbers. In Session 3, you'll learn about **string variables,** which hold anything—numbers, letters, special characters. If you get an error message like TYPE MISMATCH, you've probably tried to give a non-numeric value to a numeric variable.

Break a few rules

One of the best ways to understand a programming rule is to break it. Break every variable rule there is and see what happens. Go ahead—question authority. Here are some examples:

```
NEW
10 PRINT 1V
RUN
10
```

Your computer thought you wanted it to print a *1* and then the value of the variable V. (All variable names start with a letter.) Variables that you haven't assigned a value to automatically hold the value 0; a *1* with a *0* next to it is *10*.

```
10 PRINT = 1
RUN
?SYNTAX ERROR IN 10
```

`PRINT` is a reserved word; you can't use it as a variable.

```
10 MIMI = 5
20 MIAMI = 8
30 PRINT MIMI
RUN
8
```

Only the first two characters of a variable name really count. As far as your Apple is concerned, you assigned the value 5 to `MI` in line 10; but you changed it to 8 in line 20.

Finding variable names that are both meaningful and legal can be a bit tricky at first. So when you run into a program bug, the *first* thing you should do is check your variable names.

Summary and review

This session taught you how to use computer arithmetic and variables. You learned the rules of precedence and how to program your computer to calculate simple and then somewhat complicated arithmetic problems. You found out that variables are storage areas used to hold values and that the names you give variables should reflect the kinds of values they hold. And you saw that, like everything else in programming, there are rules for naming variables (and that breaking those rules is a great way to learn them).

Session 3

The Outside World

Up to now, all the information that went into the computer got there through your program lines. When you wanted a variable to hold some value, you used an **assignment instruction** (as in `NUMBER = 23`, so called because it *assigns* the value `23` to the variable `NUMBER`). You, the programmer, gave the program the variable's value. In this session, you'll learn how to use INPUT, an instruction that lets the program get a variable's value from the person using your program. You'll read how to construct meaningful prompting messages so your user will know what information the program needs. And you'll learn about **string variables**, which let you assign letters and special characters (not just numbers) to variables.

You'll also learn the difference between **immediate execution** and **deferred execution**, and you'll encounter new instructions that let you clear the screen (HOME) and get an updated listing of your program (LIST).

INPUT

The INPUT instruction is at the heart of **interactive programming**—programming that lets the computer and a human hold a conversation. INPUT lets you give information to your program while it's running. It makes the program wait until you (or the person using your program) types something and presses Return.

Type and run the following program; when a question mark (the INPUT prompt) appears on the screen, type a number and press Return:

```
NEW
10 INPUT A
20 PRINT A * 5
```

Your Apple computer prints whatever number you typed after the question mark. If you typed 3, your screen would look like this:

```
?3
15
```

Your computer supplies the question mark automatically.

It's just as if you had typed A = 3 as a program line. Whatever you type in response to an INPUT prompt gets assigned to the **input variable** (a variable whose value is assigned by the user, as opposed to one whose value is assigned by the programmer).

Prompts

The question mark prompts you to type something. You knew what to type (a number) because this tutorial told you. But people using your program would have a hard time knowing what to do if all they had to go on was what appeared on the screen; a question mark in itself doesn't say much.

Applesoft lets you use descriptive **prompts** to solve this problem. Prompts tell a computer user what to do next. You can use either of two ways to show what the program wants. First, you can print a line that says what to do; then use an INPUT line.

Type this program and run it:

```
NEW
10 PRINT "I had a tough night. What year is this?"
20 INPUT Year
```

Now when you run the program, the message on the screen lets you know that you need to type the year.

You can also use the INPUT instruction itself to print a prompt. A prompt with INPUT works almost like a prompt with PRINT, except that the prompt appears on the same line as the INPUT instruction:

```
NEW
10 INPUT "I had a tough night. What year is this? "; Year      INPUT and prompt
20 PRINT "Oh, great.  I thought it was "; Year + 1            New stuff here!
30 PRINT "and I missed Christmas."
```

(Be sure to give the computer an answer when it prompts you for one.) The semicolon between the quotation mark and the variable name in line 10 is important; you have to include a semicolon when you're using a prompting phrase with an INPUT instruction. Note that when you use a semicolon after an INPUT instruction, your Apple omits the question mark prompt.

❖ *Some tips on using PRINT:* Line 20 has implications you can investigate on your own. To get you started, note that:

1. There's a semicolon after the final quotation mark—the semicolon tells BASIC to show the value of the variable on the same line as the quotation.
2. Your Apple does a little arithmetic on the variable `Year`.

Here's a program that shows several examples of self-prompting INPUT lines:

```
NEW
10 PRINT "TRIVIA PROMPT GAME"
20 PRINT
30 INPUT "How many cards are in a deck? "; Cards
40 INPUT "How many U.S. congresspersons are there? "; CP
50 INPUT "How many keys are there on your keyboard? "; Keys
60 INPUT "How many days are in a leap year? "; Leap
```

❖ *Illegal names and syntax errors:* The trivia program uses descriptive variable names in all lines except line 40. The variable name CP is not very descriptive, but both Congress and Persons contain the reserved word ON. (See the list in Appendix B.) When you get a syntax error in your program and you don't know why, try changing the variable names.

More editing: adding lines

Sometimes you have to add lines to your program. If the new lines belong at the end of the program, you just type a line number larger than the last line number in the old program and start typing. But what happens if you need to add a line in the middle? Nothing to it. All you have to do is type a line number that's *between the numbers* that already exist.

For example, suppose you have the following program, and you want to include a line between lines 10 and 20:

```
NEW
10 PRINT "Remember to"
20 PRINT "the dog"
```

You want to remember to feed the dog. All you do is add the following line to your program:

```
15 PRINT "feed"
```

Go ahead and run the program. You'll see that everything turned out in the right order.

❖ *Leave intervals between line numbers:* All the sample programs you've seen in this tutorial have line numbers spaced 10 apart. If the current program had been numbered 1, 2 instead of 10, 20, you wouldn't have had room to insert the new line, and you would have had to retype the whole program.

Cleaning up with HOME

Your screen gets cluttered after you've typed and run a few programs. The HOME instruction clears the screen and places the cursor at the upper-left corner (the cursor's beginning, or *home*, position). Each time the program encounters HOME, it clears the screen and homes the cursor:

```
NEW
10 HOME
20 INPUT "HOW MANY POUNDS ARE IN A KILOGRAM?  "; LB
30 HOME
40 INPUT "HOW OLD IS THE PRESIDENT?  "; PRES
RUN
```

The screen cleared with each new question. That way there's no confusion about what the program expects, and there's no clutter from other programs.

You can also use HOME without a line number whenever you feel like doing some light housecleaning. Just type `HOME` and press Return.

Try it now:

```
HOME
```

HOME clears the screen—it *doesn't* clear memory. HOME just erases the junk cluttering your display. It has absolutely no impact on memory. (Don't confuse it with NEW.) But after you use HOME to clear your screen, you'll need a way to see your program lines again.

LIST

Type `LIST` and press return to see your program again. Try it now.

```
LIST
```

As your programs get longer, you'll use LIST more and more. Type the following program to test the different ways to use LIST:

```
NEW
10 HOME
20 PRINT "And Maud Pritchard"
30 PRINT "waddled the bible-black path"
40 PRINT "to the boat-bobbing sea"
50 PRINT "with nary a mind"
60 PRINT "for Mr. Pritchard, dead as biscuits."
```

First, run the program; then list it. Once you've listed your program, try the following variations of the LIST command to see what happens.

```
LIST 40
LIST 40
LIST - 40
LIST 20 - 40
```

Lists line 40 only.
Lists from line 40 to end of program.
Lists from beginning to line 40.
Lists from line 20 to line 40.

With the small programs you've written so far, you won't need all these variations in the LIST command. But later, when your programs are so large they roll off the top of your screen, you'll want to list small program segments.

String variables

In Session 2, you learned how to use variables with numbers. You can also use variables with text. Variables that hold text are called string variables. String variable names always end with a dollar sign ($), and you define them (that is, give them values) in nearly the same way as numeric variables:

```
NEW
10 HOME
20 Aunt$= "Aunt Lizzy"
30 PRINT Aunt$
```

When you run this program, the words *Aunt Lizzy* appear on the screen. Line 30 works the same as

```
PRINT "Aunt Lizzy"
```

You can put just about anything into a string variable. Unlike numeric variables, which accept only numbers, string variables can hold letters, numbers, symbols—even punctuation:

```
NEW
10 HOME
20 GARBAGE$= "Allof this junk-> $43$,':;"
30 PRINT GARBAGE$
```

Your computer printed everything between the quotation marks in line 20. It's important to remember that numbers are not treated as numbers when they are in string variables. They're treated as text—just symbols, a string (get it?) of characters without meaning to the computer.

Run this next program to see numbers treated at text:

```
10 HOME
20 A$ = "10"
30 B$ = "20"
40 PRINT A$ + B$
```

Instead of getting *30*, you got *1020*. The plus sign(+) doesn't "add" the string variables. (How do you add letters?) It just strings them together. In computer terms, it **concatenates** them.

You can also use string variables with INPUT. You use prompts with a string variable INPUT just as you do with a numeric variable INPUT. This next program mixes both kinds of variables:

```
10 HOME
20 INPUT "What's your name? "; NAME$
30 INPUT "Type your age: "; NUM
40 HOME
50 PRINT NAME$;
60 PRINT" is ";
70 PRINT NUM;
80 PRINT "years old."
```

Note the semicolon.

There's a space before the i and after the s.

Just to see what happens, type some letters when your Apple asks for numbers. (For example, type `eighteen` instead of the number `18`.)

As soon as you press Return, you get this error message:

```
?REENTER
```

That just means your program expected a number and got something else. Do as it says—re-enter a number (your computer wouldn't lie to you), and everything will work fine.

Variables rules recap

In case you've forgotten, here are the rules for naming variables. The last one applies only to string variables:

- A variable name must begin with a letter.
- Characters after the first one can be letters or digits.
- A name can be up to 238 characters long, but the computer recognizes only the first two. (The others are to remind you what the variable stands for.)
- Certain letter combinations (called reserved words) can't be used in any part of a variable name. See Appendix B for a list.
- All string variable names end with $.

Debugging

Murphy's law, "If anything can go wrong, it will," applies doubly to programming. (Lubarsky's Law of Cybernetic Entomology applies equally: "There's always one more bug"; but that's for a more advanced tutorial.)

Experienced **hackers** (another term for *programmers*) and beginners alike make all kinds of little errors while programming. Debugging a program (that is, ruthlessly tracking down and exterminating bugs) is a normal part of creating a computer program; more often than not, it's a major part. That's why your computer has error messages.

Knowing the difference between **immediate and deferred execution** is helpful in debugging programs. When you type RUN or NEW or LIST without a line number, the computer does what you want as soon as you press Return. This is known as immediate execution. When you write a program with line numbers, the computer defers execution until you run it. This is called deferred execution. Immediate execution is extremely useful in debugging programs.

For example, type and run the following program:

```
NEW
10 HOME
20 MONEY$ = "$1,000"
30 PRINT MONEY$
```

You get ?SYNTAX ERROR IN 20 instead of the $1,000 you expected. List line 20, and you will be in for a surprise:

```
20 M ON EY$ = "$1,000"
```

What happened to M ON EY$? It's all broken up. Type:

```
MONEY$ = "$1,000"
```

As soon as you press Return, you get a syntax error. You have a reserved word (ON) embedded in your variable name. In your program listing, you can see that ON has been separated from M ON EY$ in lines 20 and 30. You can rewrite your program with another variable name, but first test the alternate name by using immediate execution. Try the following:

```
BUCK$ = "$1, 000"
```

There was no error message this time. That means BUCK$ is acceptable as a variable name. In this case, changing the program takes only a few seconds; you've used MONEY$ only once. But consider a situation in which you've typed a much longer program, using MONEY$ 25 or 30 times—it would take quite a bit of time to change each instance of MONEY$ to BUCK$. It's a lot quicker testing out possible errors by using immediate execution than re-writing your program every time you encounter an error.

The trick to successful debugging is isolating the problem. Some error messages give you the line number where your computer detects the problem. This helps you zero in on the problem. Test the possible problem from the immediate mode as you saw in the example with MONEY$ and BUCK$. Correct the error in the program, and re-run it to see if more errors occur. If no more errors happen, then your debugging succeeded—at least as far as variable names are concerned.

You'll find more uses for immediate execution as you go along. Experimentation is the key. Try everything first with immediate execution; you'll be in for some pleasant surprises.

Summary and review

In this session, you learned that you can get information from the user with the INPUT instruction while your programs are running. Be sure to use descriptive prompts with INPUT; that way people who use your programs can know what they're supposed to type. Descriptive prompts are to the users of your programs what descriptive variable names are to you, the programmer.

You also learned about string variables. You saw that they work and look much like numeric variables, except that string variables end with $, and their values are surrounded by quotation marks in a program line.

The HOME instruction clears the screen for you. LIST lets you see all or some of the lines of the program in memory to make program debugging easier.

You also learned that you can use many programming instructions with immediate execution to help you debug programs.

Session 4

Using the Disk and Other Devices

As you write longer and better programs, you'll want to start saving them to use again. This session explains how to store programs onto disks and how to get them back again.

You'll learn about three different kinds of memory (**RAM**, **ROM**, **disk**), with emphasis on disk memory. You'll see how to store a program onto a disk with SAVE, retrieve the program with LOAD, and see a list of all the programs on a disk with CAT. You'll learn how to get rid of outdated programs on a disk by using DELETE.

You'll also learn how to use PR#1 to get a version of your program on paper instead of on the screen, and how to use PR#0 to use the screen again. And you'll end the session with a review of everything you've learned so far.

Computer memory

RAM stands for ***Random-Access Memory***. RAM is temporary. When you first turn on your computer, this memory has nothing meaningful in it. When you write a program or tell your computer to retrieve a program stored on a disk, that information goes into RAM. When you turn off your computer, all of the information in RAM evaporates.

ROM is ***Read-Only Memory***. It's a kind of memory that holds information permanently. The Applesoft BASIC language is stored in this kind of memory; when you turn your computer off, the language stays in ROM (but *not* your program). Nothing that you type gets stored in this kind of memory.

A **disk** is what you save programs on. Disk drives (the devices that disks go into) work a lot like tape recorders. With a tape recorder, you talk into the microphone, and your voice is recorded on magnetic tape. Then you rewind the tape and listen to your voice. Your computer works the same way, except that instead of using tape recorders to save what's in RAM onto tape, it uses disk drives to save information onto disks. Once you've got a program on disk, you can "play it back" again and again.

You don't have to worry about the technical details of RAM, ROM, and disks. But you'll save yourself a lot of grief if you remember that when you turn off your computer, everything in RAM disappears into electronic oblivion.

Files and catalogs

Most well-organized people put written records in files so they can find the records again. So too with computer records. Programs stored on disk are also called **files**. There are several other kinds of files, but the only kind you have to know about for now are **program files**—the name given to programs stored on disks.

Making a list or **catalog** of what files are stored in a file cabinet makes it easier to locate a file when you need it. Essentially, that's what your computer does when you save a program on a disk. You store your program by using the SAVE command, and the name of the program is placed in a catalog. When you want to use a program, you look it up in the disk's catalog with the CAT command to make sure it's there; then you retrieve it by using the LOAD command.

❖ *Commands versus instructions—a matter of terminology:* That last paragraph used the term *command* several times. A command is like an instruction in that it tells the computer to do something. The difference between a command and an instruction lies almost entirely in when the computer does what you want. Essentially, a command is an order that the computer executes immediately; an instruction is an order whose execution is deferred. It's just a matter of terminology.

How to save programs

Storing a program onto a disk is the easiest thing in the world. You issue the SAVE command, giving your program a name you can use later to get it back from the disk.

To get some practice, first type in this program:

```
NEW
10 PRINT "This is my very first saved program."
20 PRINT "I'm very proud of it"
30 PRINT "(or I will be, if I can get it back)."
```

Now you need to think of a name. Here are the rules for naming a program.

- A program's name can be up to fifteen characters long.
- The name must begin with a letter.

- You can use letters, digits, and periods in the filename, but you can't use any other characters, and you can't include any spaces. You can use both uppercase and lowercase characters, but the computer converts all letters to uppercase.
- All filenames on a given disk must be unique. But *all* characters in the name count, not just the first two, and you don't have to worry about reserved words. So coming up with different filenames shouldn't be much of a problem.
- The name should reflect what the program does.

Here are some legal filenames:

CHECK.BOOK

ADDING.PROGRAM

AH.1ANDAH.2

NOT.4.SALE

These names, though, are *illegal:*

Illegal Name	Problem
1ONE	Begins with a number.
THIS.PROGRAM!	Exclamation mark is illegal.
.POINT	Begins with a period.
A.REALLY.TRULY.NIFTY.PROGRAM	Too, too, long.
GREAT STUFF	There's a space.

(Many people use periods in filenames where they'd use spaces if they could.)

Save your program onto a disk now. You can use whatever legal name you want; MY.FIRST.FILE seems like an appropriate one.

Type this line and press Return:

```
SAVE MY.FIRST.FILE
```

The disk whirs and kerchunks a bit. When it stops, a copy of your program is safely stored on the disk. Note that word—copy. Storing a program on disk doesn't have any effect on what's in the computer's memory.

Type `LIST` and press Return; you'll see that the program is still there.

Reading the catalog and retrieving a program

Once you've saved your program to the disk, type NEW and press Return. Now you know for sure that there's nothing in memory. (Type LIST and press Return to see for yourself.)

To look at the files on your disk, use the CAT command. You'll get a list of all the files on the disk.

Type this command and press Return:

```
CAT
```

Assuming there are no other programs on the disk, your screen will look like this:

```
]CAT

/PRACTICE

NAME              TYPE       BLOCKS      MODIFIED

MY.FIRST.FILE      $08          33      <NO DATE>

BLOCKS FREE      240          BLOCKS USED:     40

]
```

(Of course, your screen will look different if the disk already has other programs on it.) The program MY.FIRST.FILE is now in the catalog. (For information on what the rest of the display means, see the manual that came with your computer.) The next step is to retrieve the program. To do that you need a new command, LOAD.

Type this command and press Return:

```
LOAD MY.FIRST.FILE
```

You'll hear your disk drive whir a second, and then the prompt and cursor will reappear. That means your program was successfully loaded into memory.

To make sure it's the program you saved, list it:

```
LIST
```

Your program appears, just as it was when you saved it.

❖ *LOAD does a NEW:* When you load a program, your computer first clears its memory of any program that might already be there. This means you don't have to worry about two programs being mixed together. (It's possible to combine two programs, but the technique is too advanced for this tutorial.) Think of LOAD as having an automatic NEW attached to it.

Cleaning up

If you're really careful when you write programs, you'll save different versions as you go along. For example, you might have saved these programs on your disk:

STAMPS.V1
STAMPS.V2
STAMPS.V3

If you know for sure that the last version of your program, STAMPS.V3, is the only one you plan to use, you might as well get rid of the other versions and free up room on your disk. You delete files by using the DELETE command.

To delete STAMPS.V1, type

```
DELETE STAMPS.V1
```
Press Return.

You'll hear the disk whir, and STAMPS.V1 will be just a memory (human, not computer). Just think of DELETE as the opposite of SAVE, and use the same format.

❖ *DELETE's not reversible:* DELETE is forever. Once you delete a program from the disk, it's gone. Be sure that you want to get rid of a program before you use DELETE.

For printer owners: printing your listings

So far, you've sent your program to the screen and to the disk. You can also send your program (and anything else you type) to the printer.

Printing out a program, especially a long one, is *extremely* helpful in program debugging; your experience will show you how very true this is.

To list a program on your printer, follow these steps:

1. Make sure your printer is properly connected to the computer.
2. Check that you have paper properly loaded.
3. Be sure the printer is turned on.
4. Type `PR#1` and press Return.

(If you don't follow any one of the first three instructions, your computer will appear to be stuck.) The PR#1 command makes everything that would go to the screen go to the printer. If you type LIST after you've typed a PR#1 command, your printer will clank out the listing (unless you've typed LIST incorrectly—in which case the syntax error message gets printed).

To see the computer's output on your screen again and to stop using your printer, type this:

```
PR#0
```

and press Return. The command will appear on the printed page; but after that, subsequent commands and listings will appear on the screen instead.

Bugs can be tough to find in longer programs, especially when your listing is so long that it scrolls off the screen. Printing out your listings can save a great deal of debugging time.

Type this program and try listing it on your printer:

```
NEW
10 HOME
20 PRINT "This program will be listed to my printer."
30 PRINT "If there's a bug here, the printer"
40 PRINT "will help me track it down."
PR#1
LIST
```

Your printer gives you a **hard copy** listing of the program.

Before you turn off your printer with PR#0, run the program to see what happens. Then type PR#0 to get your BASIC prompt (]) back on your display screen.

Using what you've learned

You've had less to learn in this session than in the three previous ones. Use your remaining BASIC study time to write some programs that use all the instructions and operators you've learned so far. Here's a list to jog your memory:

Instructions

HOME	INPUT	PRINT

Operators

+	–	*
/	(	)

Commands

CAT	DELETE	NEW
LIST	LOAD	PR#1
PR#0	RUN	SAVE

Concepts

Immediate and Deferred Execution	Line Numbers with Intervals
Meaningful Names	Numeric Variables
Precedence	Prompting Messages
String Variables	

Summary and review

In this session, you learned how to store programs onto disks by using the SAVE command, and how to get them back by using LOAD. You learned how to name programs, and which characters are legal in a name and which ones aren't. You saw that CAT gives you a list of all the files on your disk, and that if you use PR#1, whatever ordinarily goes to the screen goes to the printer. (PR#0 sends information to the screen again.)

Session 5

Loops and Conditions

In the first few sessions, you learned the rudiments of BASIC programming. Now it's time to get down to some more advanced stuff. In this session you're going to learn about three very powerful principles: loops, relationals, and conditionals. You'll also read about some BASIC short cuts that make programming easier, and you'll learn some other helpful instructions.

Loops

To **loop** is to go over the same part of a program more than once. For example, suppose you want to get ten names with INPUT and print them one after another onto the screen. It would be a lot easier to repeat the part of the program with the **INPUT** instruction than to write ten separate lines with INPUT:

```
NEW
10 HOME
20 INPUT "Gimme a name: "; NAME$
30 PRINT NAME$
40 ...
```

How do you get back to line 20?

What you need is some instruction that lets your program loop back to line 20 to get another name. That instruction is GOTO.

GOTO

The GOTO instruction directs the program to go to any line you name. This program clears the screen, then skips to (or **branches** to) line 40 instead of going to line 30:

```
NEW
10 HOME
20 GOTO 40
30 PRINT "Hey!  I thought I was next!"
40 PRINT "I'm the only line you'll see!"
```

This never gets printed!

Here's another example. Type the first program of this session, but this time type

```
40 GOTO 20
```

for the last line. Then list the program. It should look like this:

```
10 HOME
20 INPUT "Gimme a name: "; NAME$
30 PRINT NAME$
40 GOTO 20
```

This program repeatedly asks for a name and then prints out what you type. The program will go on doing this forever as long as somebody keeps typing in names (or until somebody pulls the plug); every time the program reaches line 40, it goes back to line 20.

❖ *Infinite loops:* What you've got here is an infinite loop. Sometimes, infinite loops can be helpful—this isn't one of those times. To get out of the loop before you run out of names (or patience), press the keys marked Control and C at the same time; release them and press Return. That's called pressing Control-C; you'll run across this term often if you read computer books and magazines. When you press Control-C, your computer will announce:

```
BREAK IN 20
```

The message means that you "broke into" the program at line 20. When a program gets stuck (or hangs), sometimes the only way to regain control is with Control-C.

This program solves the problem of getting lots of names without retyping INPUT lines again and again. But it's out of control. You need a way (other than Control-C) of getting the program to stop looping when you've had enough.

Conditional branching with IF...THEN

BASIC has a two-part instruction called IF...THEN. It gives your program the power to make decisions—which, as it turns out, is just what you need to solve the infinite loop problem. The general format of IF...THEN looks like this:

IF <*something is true*> THEN <*perform some action*>

An IF...THEN instruction decides whether or not something is true. If what you say in the first part between the words *IF* and *THEN* (called the **condition**) is true, then your computer does whatever you put after *THEN*. If the condition is *not* true, then the program ignores everything after *THEN* and drops to the next line.

To see this in action, add two lines to your infinitely looping program:

```
25 IF NAME$ = "enough" THEN GOTO 50
50 PRINT "And that ends the name list."
```

Here's the whole listing:

```
10 HOME
20 INPUT "Gimme a name: "; NAME$
25 IF NAME$ = "enough" THEN GOTO 50
30 PRINT NAME$
40 GOTO 20
50 PRINT "And that ends the name list."
```

Run the program now; after you've typed a few names, type `enough` and the program ends.

Building on the model

Going by the model IF *<something is true>* THEN *<perform some action>*, in the previous example the something-that's-true (the condition) is *NA$ = "enough."* When NA$ was anything *except* "enough," the program went on looping; when it was enough, the program branched to the final line. The branching was the perform-some-action part.

- *Start a saving plan:* As you type in your programs, you should get into the habit of saving them to your disk before you run them. Then, save them often as you develop and change them—once every ten minutes or so will do nicely. There'll be situations when even Control-C won't get you out of trouble (like, for instance, when your little brother playfully flicks off the power switch). If you save the program often, you won't have to recreate and retype your latest refinements.

Relational operators

Here are some more examples of IF...THEN instructions. Pay careful attention to the conditions; you'll see some symbols you haven't seen before:

```
IF NA$ = "QUIT" THEN GOTO 100
IF A$ <> "APPLE" THEN PRINT "YOU LOSE!"      <> means "not the same as".
IF SUM > 10 THEN X = 50                       > means "greater than".
IF COUNT< 100 THEN GOTO 20                    < means "less than".
```

Those little angle brackets are called **relational operators.** They describe a *relation* that exists between two things. Here's a chart that shows all the relational operators and what they mean:

Operator	Meaning
>	greater than
<	less than
=	equal to
<>	not equal to
>=	not less than
<=	not greater than

The next two programs give you some examples of what you can do with relationals, GOTO and IF...THEN instructions. They also present you with some challenges, teach you a new instruction or two, and give you a few BASIC short cuts.

Comparing Values: This program asks you for two numbers, then tells you which number is the lower one. The program has a few surprises in it to keep you from getting bored.

First, type the program. Then see if you can figure out what's going on *before* you run it. Finally, run it and see if you were right.

```
NEW
10 HOME
15 PRINT "To end the program, type a 0 for the first number."
20 INPUT "Enter the first number:  "; N1
25 IF N1 = 0 THEN END
30 INPUT "Enter the second number:  "; N2
35 IF N1 > N2 THEN GOTO 100
40 IF N1 < N2 THEN GOTO 200
45 PRINT "Those numbers are the same!"———How does this work???
50 GOTO 20
100 PRINT N2; " is lower than "; N1
110 GOTO 20
200 PRINT N1;" is lower than "; N2
210 GOTO 20
```

Here are some questions for you to consider before reading further:

1. There's a new instruction in line 25—END. What does it do?
2. Line 45 will print its message only if both numbers you type for lines 20 and 30 are the same. Why?

How The Program Works: Line 15 lets you know what to do to stop the program without using Control-C. The END instruction in line 25 does the work of stopping the program—but only if you type a 0. Line 45 is executed only when values for N1 and N2 are the same. To see why, look at the two previous lines. Line 35 goes to one part of the program if N2's value is lower than N1; line 40 goes to another part of the program if the opposite is true. Being the literal "thinker" that it is, your computer continues on to the next line (line 45) only if there's no reason not to—in this case, if both values are the same.

Assigning Variables: This next program shows how IF...THEN can assign different values to variables. In this instance, the values are different words. (They could just as well be numbers.)

Type the program. Before you run it, figure out

1. What are all those question marks for?
2. What's strange about line 80?
3. What's line 80 for, anyway?

Be sure to figure out the challenges before you run this zoologically questionable program:

```
NEW
10 HOME
20 ? "1. SWIMS"
30 ? "2. WALKS"
40 ? "3. FLIES"
50 PRINT
60 PRINT "Think of an animal.  Then choose a"
65 ? "number that best describes how your"
70 INPUT "animal moves. "; NUMBER
80 IF NUMBER> 3 THEN 10
90 IF NUMBER < 1 THEN 10
100 IF NUMBER = 1 THEN ANIMALS ="Fish"
110 IF NUMBER = 2 THEN ANIMALS = "Mammal"
120 IF NUMBER = 3 THEN ANIMALS = "Bird"
130 PRINT
200 PRINT "I bet your animal is a "; ANIMALS
```

Those question marks are a short-hand way of typing PRINT. Saving four keystrokes each time you want to use PRINT can save you lots more time than you think. When you list your program, each question mark will be converted to PRINT.

Line 80 is peculiar in that it leaves out the word GOTO. It turns out that any of the following forms work for the GOTO instruction within an IF...THEN:

IF NUMBER > 3 THEN GOTO 10
IF NUMBER > 3 THEN 10
IF NUMBER > 3 GOTO 10

In other words, you can omit THEN or GOTO—but not both.

The purpose of lines 80 and 90 is to set traps to make sure anyone using the program doesn't put in a number that's beyond the range of choices. Traps give your users another chance in case they make a mistake (which is an annoying human tendency).

Use REM for remarks

The REM instruction lets you write notes to yourself about what your program does, and lets you include the notes in the program. These notes show up only when you list your program; people can't see them when they run it.

For example, you can use REM instructions to keep information about the program handy, or to tell you what the program segment is doing:

```
100 REM*******************************
110 REM        The Great American Computer Program
115 REM        by Throckmorton Scribblemonger
120 REM        Version 16.5
125 REM        July 4, 1987
130 REM        *******************************
135 REM                       Clear the screen
140 HOME
145 COMMENT$ = "REM comments don't appear on the screen."
150 REM                       Print a message on the screen
155 PRINT COMMENT$
160 _
```

REM instructions are reminders for people, not for computers. REM instructions do nothing to your program. When the program reaches one, it ignores the REM instruction (and anything after it on the same line) and goes on to the next line.

❖ *Put program name in a REM line:* Make the first or second line of your program a REM line containing the program's name. Then, when you change the program and want to save the new version onto a disk, you'll always know what name to use.

Practice time

You covered a lot of ground in this session. Before going on, experiment with what you've learned. Go back and change the example programs. Try to "break" some programs; find the limits of the instructions you learned in this session. Certainly write some programs of your own. Make mistakes—they're free.

Summary and review

This session showed you how your computer can loop and make decisions (that is, process information). You use loops to repeat a process several times. Instead of having to repeat the same line throughout your program, you can use GOTO to repeat the lines. This saves a lot of time in building your programs.

The IF...THEN instruction is your computer's "decision maker." With IF...THEN, you can branch to different options and jump out of infinite loops. You can trap mistakes with IF...THEN to make sure the person using your program types information for INPUT instructions within the program's range.

You learned some short cuts for writing GOTO instructions within IF...THEN instructions, and you saw how to use the question mark in place of PRINT.

Finally, you saw how to use REM to remind yourself what a particular part of your program does. By using REM throughout your programs, you can clearly organize your program lines; by marking program segments to make them easier to find, you make debugging easier.

Session 6

Graphics

Up to this point, all you've seen on your Apple computer is text. But you can also produce some wonderful color graphics. Your computer has several graphics modes; in this session you'll learn the one called **low-resolution graphics**. (It's the easiest to use.)

You'll learn the difference between your computer's text and graphic modes, while learning the GR and TEXT instructions. You'll see how to use COLOR= to set one of sixteen colors to use with PLOT (for plotting points), VLIN (for drawing vertical lines) and HLIN (for drawing horizontal lines).

Besides all this, you'll learn the RND instruction for producing random numbers—which you'll use, in turn, to produce some pretty snazzy graphics.

Text and graphics

Your computer has separate **modes** for text and graphics. (A **mode** is any of several ways a computer interprets information.)

To get started, you need to know two instructions—one to get into graphics mode and one to get out. When you turn on your computer, it automatically goes into text mode. When you type the instruction GR for graphics, your computer goes into graphics mode.

Type the command

```
GR
```

(don't use a line number) and press Return.

Your screen went blank, and the cursor popped up at the bottom of the screen. The top of the screen, above the cursor, is for graphics; it takes up 20 of the screen's 24 lines. The bottom four lines are for text.

❖ *For non-color users:* Everything in this session assumes you're using a color monitor or color television set. If you're using a black and white TV or a monochrome monitor, the shapes you draw are displayed in different patterns instead of in colors.

Type and run this program:

```
NEW
10 GR                                             Turn on graphics
20 COLOR= 3                                       Pick a color.
30 PLOT 19, 19                                    Plot a point.
100 PRINT "Purple Square on Black Field (1986)"   Great art always has a title.
```

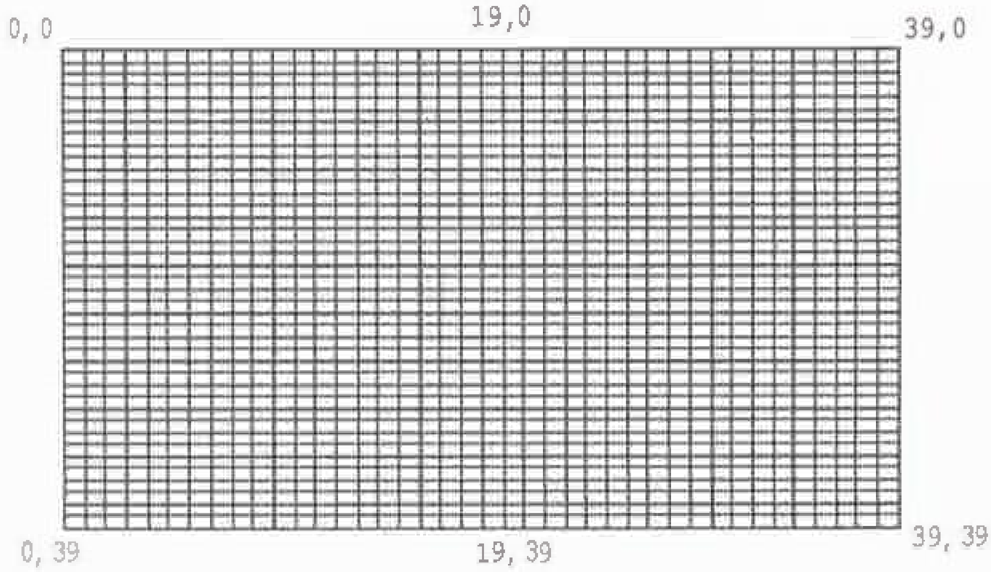
A 40-by-40 canvas

The low-resolution graphics screen is a 40-by-40 grid. The PLOT instruction places a block in the horizontal and vertical positions you specify. PLOT 0, 0 would place a block in the upper-left corner, and PLOT 39, 39 would put a block in the lower-right corner.

Add the following lines to the program you just typed to see the limits of PLOT. (Don't worry about not being able to see the rest of your program; you'll see it all again in a minute.)

```
40 PLOT 0, 0
50 PLOT 39, 39
RUN
```

When you run it now, you'll see three blocks running diagonally down the screen. The one in the upper-left corner is position 0, 0; the one at the lower-left is position 39, 39. Here's what the whole matrix looks like:



Before continuing, see if you can plot blocks in the lower-left corner, upper-right corner, and the middle of the left and right sides. Once you know how to do that, you can plot anywhere you want.

Seeing your listing again

When you added new lines to your program, all the text above the new lines scrolled out of sight behind the graphics. To see your listing again, you'll need to get back to text mode.

Using immediate execution (that is, giving an instruction without a line number), type :

```
TEXT
```

and press Return.

The strange pattern you see is the result of your Apple looking at its own graphics symbols and interpreting them as text. To humans, it's just junk (or Punk Art). Type:

```
HOME
```

and press Return. Then list the program. If you got the last program right, your listing looks something like this:

```
LIST
10 GR
20 COLOR= 3
30 PLOT 19, 19
40 PLOT 0, 0
50 PLOT 39, 39
60 PLOT 0, 39
70 PLOT 39, 0
80 PLOT 0, 19
90 PLOT 39, 19
100 PRINT "Purple Square on Black Field (1986)"
```

Plotting colors with COLOR=

The COLOR= instruction (the = is part of the instruction) lets you decide what colors go where. Here's a chart of all the colors you can use:

Number	Color	Number	Color
0	Black	8	Brown
1	Magenta	9	Orange
2	Dark Blue	10	Dark Gray
3	Purple	11	Pink
4	Dark Green	12	Green
5	Gray	13	Yellow
6	Blue	14	Aqua
7	Light Blue	15	White

The COLOR= instruction by itself won't add color to anything. It colors only what you draw on the screen. The color you set with COLOR= stays in force until the next COLOR= instruction.

Add this new line to your program:

```
65 COLOR=13
```

Now run the program and see what happens.

❖ *Uncluttering the text:* You've got only four lines of text when you use graphics mode; you don't need to have one of those four lines cluttered up with a left-over RUN instruction. Aesthetics are, after all, important. Adding a HOME instruction early in the program (say, at line 5) will take care of the problem nicely.

Using variables for plotting and coloring

You can use variables for plotting points and setting colors. Instead of using absolute numbers as in COLOR= 10 or PLOT 10, 20, you can type COLOR= BLUE or PLOT COLUMN, ROW.

Type this next program. Before you run it, see if you can figure out what's happening:

```
NEW
10 GR
20 COLOR= 11 ------------------------ 11 is pink.
30 PLOT COLUMN, ROW ----------------- Initial value of all variables is zero.
40 COLUMN = COLUMN + 1 -------------- Don't panic; explanation to follow.
50 IF COLUMN > 39 GOTO 80 ----------- 39 is highest column number on grid.
60 ROW = ROW + 1
70 GOTO 30 -------------------------- Do it all again.
80 END
```

Incrementing columns and rows

Line 40 is called a **counter** in computer terms. Every time the computer executes line 40, the value of the counter (called COLUMN) increases by one. In everyday language, the line says, "Take the old value of COLUMN and add 1 to it. From now on, use the new value." The original value of COLUMN is 0 (all variables start with a value of 0). After the computer passes line 40 the first time, COLUMN holds 1; after the second time, it holds 2. And so it goes until COLUMN holds a value greater than 39 (according to line 50), and the program ends.

❖ *For budding computer geniuses only:* Draw a diagonal line that crosses the first one—that is, one that starts at the upper-right corner and goes to the lower-left. It's tougher than it sounds, but once you figure it out, its simplicity will astound you.

Maybe.

Hint: Start at 39 and work backwards.

Drawing horizontal and vertical lines

The PLOT instruction creates one block at a time. To draw a vertical or horizontal line with PLOT, you could program a sequence of connected blocks, just as you did to make a diagonal line. With Applesoft BASIC, though, it's a lot easier to use HLIN (for *horizontal line*) and VLIN (for *vertical line*). You use the same plotting coordinates as with PLOT. For HLIN, you put in the beginning and ending *horizontal* positions at a vertical position with the AT instruction:

```
HLIN FIRST, LAST AT ROW
```

For VLIN, give the beginning and ending vertical positions at a horizontal position:

```
VLIN FIRST, LAST AT COLUMN
```

Look at this next example to see how to make a cross on the screen:

```
NEW
10 GR
20 COLOR = 15
30 HLIN 10,30 AT 19 ———————— Draws a line left to right
40 VLIN 10,30 AT 19 ———————— Draws a line up and down.
```

Lines 30 and 40 look identical except one uses HLIN and the other VLIN.

As an exercise, change line 30 so that instead of a cross, the lines make a T with the horizontal line right across the top of the vertical line.

A universal line-drawer

This program lets you put in different values to draw different lines. Use it until you get a feel for where different values draw lines on the matrix:

```
NEW
5 TEXT
10 HOME
20 INPUT "Beginning block of HLIN: "; HB
30 INPUT "Ending block of HLIN: "; HE
40 INPUT "Row for HLIN: ";HP
50 INPUT "Beginning block of VLIN: ";VB
60 INPUT "Ending block of VLIN: ";VE
70 INPUT "Column for VLIN: ";VP
100 REM ********************
110 REM DRAW THE LINES
120 REM ********************
130 GR
140 COLOR= 15
150 HLIN HB,HE AT HP
160 VLIN VB,VE AT VP
170 INPUT "More lines (Y/N)?"; AN$
180 IF AN$ = "Y" THEN 10
```

Try different values until you can predict exactly where the vertical and horizontal lines will go. Just for the experience, enter values beyond the range of the matrix (that is, greater than 39). For example, enter a value of 50 to see what happens. Learning what error messages mean is just as important as learning how to do things *without* getting error messages. Later, when you make a mistake (and everybody makes mistakes while learning to program), you'll have a better idea of how to fix it.

Before you go on, modify the program so that it asks you what color you want to use. If you're *really* feeling on top of things, add some code that displays the line coordinates at the bottom of the screen; the resulting text should look like this:

```
Horizontal line from 10 to 35 in row 15
Vertical line from 18 to 26 in column 25
```

Random graphics

Your computer has a random-number generator built into it. With it, you can have your computer pull numbers out of its electronic hat. The RND instruction by itself generates random decimal numbers between 0 and 1.

Try this program:

```
NEW
10 TEXT
20 HOME
30 COUNT = COUNT + 1
40 PRINT RND( 1)
50 IF COUNT = 5 THEN GOTO 70
60 GOTO 30
70 END
```

Last program used graphics.
Clear the junk.
Add one to counter.
Print a random number.
Five numbers printed yet?
If no, get another random number.
If yes, then end.

RND always prints a decimal number between 0 and 1. But by multiplying whatever it produces by some whole number, you can make it cough up numbers your computer can use to make graphics.

Change line 40 to this:

```
40 PRINT RND( 1) * 40
```

Parentheses after RND required.

Now run the program again. All the numbers are greater than 0 and less than 40.

❖ *Parentheses required with RND:* You must follow RND with a number enclosed in parentheses. To make sure RND produces a different series of random numbers every time you use it, use 1 or a higher number. (Experimentors: to get a repeating series of numbers, use 0 or a negative number.)

Type and run this variation on the same program; it puts each random number in a variable as the random number is produced:

```
5 TEXT
10 HOME
20 NUMBER= RND( 1) * 40
30 PRINT NUMBER
40 IF NUMBER> 38 THEN GOTO 60
50 GOTO 20
60 PRINT "That's it!"
70 END
```

This program runs until the random-number generator produces a number greater than 38. Sometimes it lists a lot of numbers, and other times just a few, depending on how soon a number greater than 38 comes up. Notice, by the way, that the program generates numbers between 0 and 39.9999—never any number as high as 40.

All you do to generate random graphics is to use randomly generated variables in PLOT. You can also use randomly generated numbers to produce different colors as well.

Type and run this next program for some colorful results:

```
10 GR
15 REM COLORS 0 - 15
20 HUE = RND( 1) * 16
25 REM HORIZONTAL VALUES 0 - 39
30 COLUMN = RND( 1) * 40
35 REM VERTICAL VALUES 0 - 39
40 ROW= RND( 1) * 40
50 COLOR = HUE
60 PLOT COLUMN, ROW
70 IF ROW> 39 THEN END
80 GOTO 20
```

❖ *What about the fractional part?* A graphics instruction looks only at the whole part of a number; it ignores the fractional part. To a graphics instruction, 39.999999 is 39; 1.111111 is 1; and any positive number less than 1 is 0.

A Minor Challenge for You: Nothing heavy—just change the program so that it randomly generates horizontal and vertical lines of random length.

Summary and review

Color graphics add another dimension to your programming. You can create useful programs with them, and they're lots of fun to play with. Low-resolution graphics make rough figures, but they have a lot of color and make good graphs. You use PLOT, HLIN, VLIN, and COLOR= along with other programming instructions to build graphic images. The random-number generator inside your Apple can automatically churn out any range of numbers you want. When you combine RND and the graphics instructions, you can create a kaleidoscope of shapes and colors.

Session 7

Controlled Loops

In this session, you'll continue to learn about loops. You already know how to do loops with GOTO. Here, you'll learn about the FOR\NEXT instruction, which lets you decide in advance how many times a loop gets executed. You'll learn some tricks using loops (like how to slow down program execution). And as a bonus, you'll see how to do simple animation.

The session ends with a list of all the commands, instructions, operators, and programming concepts you've learned so far; the list is impressive.

FOR\NEXT

You saw in Session 5 how to use a counter with IF...THEN to control how many times your computer performs a loop:

```
NEW
10 GR
20 COLOR= 11
30 PLOT COLUMN, ROW                 Loop starts here.
40 COLUMN = COLUMN + 1              Here's the counter ...
50 IF COLUMN > 39 GOTO 80           ... to get you out of the loop.
60 ROW = ROW + 1
70 GOTO 30                          Loop ends here.
80 END
```

The FOR\NEXT instruction lets you define at the outset how many times your program will loop. It has its own built-in counter. Here's the structure of this two-part instruction:

FOR < *variable*> = < *start* >TO < *finish*>
< *instructions in here get carried out*>
NEXT < *variable* >

This program uses FOR\NEXT to repeat a loop 10 times. Type and run it:

```
NEW
10 TEXT                               Last program was graphics; restores text mode.
20 HOME                               Clears away the junk.
30 FOR ROUND = 1 TO 10                This is the FOR part ...
40 PRINT "This is round # "; ROUND    ...all instructions within the loop get executed...
50 NEXT ROUND                         ...and this is the NEXT part.
```

When you run this program, the value of ROUND goes from one to ten. The variable ROUND behaves just like any other variable, and as you see on your screen, the numbers represent the values the loop generates. All of the lines between the FOR and the NEXT are repeated until the loop reaches its maximum value. In this case that value is ten.

You can start the loop at any value you want. Here's a bunch of line 30's you can substitute (one at a time, of course) to see what happens:

```
30 FOR ROUND = 0 TO 20
30 FOR ROUND = -10 TO 10              Begin with a negative number.
30 FOR ROUND = 128 TO 255
```

Instead of using numbers to set up the FOR\NEXT loop, you can use variables. For example, the following program lets you use INPUT to set up the beginning and ending values of the loop:

```
NEW
10 HOME
20 INPUT "Lowest number: "; LOW
30 INPUT "Highest number: "; HIGH
40 HOME
50 FOR NUM = LOW TO HIGH
60 PRINT NUM
70 NEXT NUM
```

The FOR\NEXT loop works equally well with graphics. By setting up a FOR\NEXT loop, you can draw diagonal lines to go with your vertical and horizontal ones. Here's the original program:

```
10 GR
20 COLOR= 11                         Loop starts here.
30 PLOT COLUMN, ROW
40 COLUMN = COLUMN + 1               Here's the counter ...
50 IF COLUMN> 39 GOTO 80             ...to get you out of the loop.
60 ROW = ROW + 1
70 GOTO 30                           Loop ends here.
80 END
```

Here's the FOR\NEXT version:

```
10 GR
20 COLOR= 11
30 FOR COUNT= 0 TO 39
40 PLOT COUNT, COUNT
50 NEXT COUNT
```

Using STEP with FOR\NEXT

Sometimes you'll want to count backwards or skip numbers in a program. Use STEP with FOR\NEXT to specify the direction of the count and the increment.

For example, this program counts by 5's. Type and run it:

```
NEW
5 TEXT
10 HOME
20 FOR NUMBER = 10 TO 100 STEP 5     Here's the line to look at.
30 PRINT NUMBER
40 NEXT NUMBER
```

And this one counts backwards:

```
10 HOME
20 FOR COUNTDOWN = 10 TO 0 STEP -1
30 PRINT COUNTDOWN
40 NEXT COUNTDOWN
50 PRINT "BLAST OFF!"
```

(Five extra points if you can draw the rocket.)

You can even create simple animation that uses forward and backward stepping in graphics. Here's a bouncing block:

```
NEW
10 GR
20 FOR BOUNCE = 0 TO 39
30 COLOR = 15                      Sets color to white...
40 PLOT 19, BOUNCE                 ...so you can see the block.
50 COLOR = 0                       Sets color to black...
60 PLOT 19, BOUNCE                 ...so you can erase it.
70 NEXT BOUNCE
100 REM ************
110 REM BOUNCE UP
120 REM ************
130 FOR BOUNCE = 39 TO 0 STEP -1
140 COLOR= 15
150 PLOT 19, BOUNCE
160 COLOR = 0
170 PLOT 19, BOUNCE
180 NEXT BOUNCE
```

You can see how easy that was to do with a backward STEP. By the way, the ball will keep on bouncing if you add :

```
190 GOTO 20
```

It'll get really pretty if lines 30 and 140 read:

```
COLOR= RND( 1) * 16
```

To make the ball bounce diagonally, change—well, you figure that out on your own.

Delay loops

Sometimes you'll want to slow down your program so that you can see things happen on the screen that ordinarily go by too fast.

For example, type and run this next program to print a message on the screen, clear the screen, and print another message:

```
NEW
2 TEXT
5 STALL= 1000                                  Change this value to change the pause length.
10 HOME
20 PRINT "A VERY IMPORTANT MESSAGE"            Show message...
30 FOR PAUSE=1 TO STALL                        ...hold it...
35 NEXT PAUSE
40 HOME                                        ...clear the screen ...
50 PRINT "BE SURE TO SAVE YOUR PROGRAMS"       ...show message...
60 FOR PAUSE=1 TO STALL                        ...hold it...
65 NEXT PAUSE
70 HOME                                        ...clear the screen.
80 PRINT "BEFORE YOU TURN OFF YOUR COMPUTER!"
```

The empty FOR\NEXT loops between showing the messages and the HOME instructions give you time to read what's on the screen. (Take out lines 30, 35, 60, and 65— just type their line numbers and press Return—and the messages will fly by too fast for you to read when you run the program.)

Use **delay loops** when you want several messages to be presented automatically, and when you don't want to press any keys to see the next message. You can make flashcard-type review programs with short delay loops.

For a spelling quiz, have a word pop on the screen long enough to be read but not long enough to be spelled. Here's a quick one to try:

```
NEW
5 STALL = 150 ———— Change this value to change the pause length.
10 HOME
20 REM *******************
30 REM SPELLING WORDS
40 REM ******************
50 A$ = "DUCK"
60 B$ = "JEWELRY"
70 C$ = "PROGRAMMING"
100 REM ***************
110 REM SPELLING TEST
120 REM ***************
130 PRINT A$
140 FOR LOOK = 1 TO STALL ———— Here's a delay loop.
145 NEXT LOOK
150 HOME
160 INPUT "SPELL THE WORD "; SPELL$
170 IF SPELL$ = A$ THEN RIGHT = RIGHT + 1 ———— Counter adds up correct spellings.
180 PRINT B$
190 FOR LOOK = 1 TO STALL ———— Another delay loop.
195 NEXT LOOK
200 HOME
210 INPUT "SPELL THE WORD "; SPELL$
220 IF SPELL$ = B$ THEN RIGHT = RIGHT+1
230 PRINT C$
240 FOR LOOK = 1 TO STALL ———— Yet another delay loop.
245 NEXT LOOK
250 HOME
260 INPUT "SPELL THE WORD "; SPELL$
270 IF SPELL$= C$ THEN RIGHT= RIGHT +1
280 HOME
290 PRINT "You got "; RIGHT; " words right."
```

You can change the values in the delay loop (line 5) to give yourself more or less time to see the word.

A quick review

You've come a long way in programming already, so now would be a good time to review what you've learned in these first seven sessions. In general, it's important to keep things simple—take programming a little chunk at a time. Here's a list of everything you've learned so far. If you've forgotten any of these terms, look them up in the glossary or check the index and go back to the appropriate session to read about them again:

Commands

CAT	DELETE	NEW
LIST	LOAD	PR#1
PR#0	RUN	SAVE

Instructions

COLOR=	END	GR
HOME	FOR[STEP]\NEXT	GOTO
HLIN	IF ... THEN	INPUT
PLOT	PRINT	RND
REM	TEXT	VLIN

Operators

+		*
/	(	)
<	>	=
>=	<=	

Concepts

Counter	Delay Loops
Immediate and Deferred Execution	Line Numbers
Loops	Meaningful Names with Intervals
Numeric Variables	Precedence
Prompting Messages	String Variables

Experiment before you continue

The final three sessions give you some refinements on the instructions and techniques you've learned so far, and introduce some more tricks and techniques. Before you go on, use what you've learned to invent your own programs and to experiment. It's important to enjoy what you do with your computer, and by writing programs that do things you like, not only will you learn programming, but you'll have a good time as well.

Summary and review

In this session you worked with loops again—but these were controlled loops. You refined your use of counters and discovered a new loop called FOR\NEXT. You learned something about computer animation, and you saw how to slow down a program by using delay loops. Then (unless you took this opportunity to challenge authority) you went over all the instructions and concepts you've learned so far, and you created new programs of your own design.

Session 8

Programming With Style: Modular Programming

You have enough knowledge now to write some very useful programs. In fact, at the end of this session, you'll be assigned the task of constructing a program to balance a checkbook.

Notice the word *constructing* in that last sentence. The best programs aren't just lists of code lines; rather, they're well-planned collections of program segments, each segment with its own job. In this session, you're going to learn about program organization and the concept of program modules.

GOSUB\RETURN

You'll often want to do the same thing in different parts of a program. For example, in Session 7 you used the same delay loop three times in a fairly short program:

```
60 FOR PAUSE = 1 TO STALL
70 NEXT PAUSE
```

Imagine a program in which you used the same lines ten, twenty, or thirty times—and how tiresome typing the same thing again and again would become (and how much of your computer's RAM your program would use). Now consider the more common situation, where the repeated **routine** (that is, collection of lines that does one specific function), rather than being just four lines long, is 10 or more lines long. By the time you were finished, you'd wear your fingers down to the second knuckle.

BASIC's GOSUB\RETURN instruction is made for just such situations. You type a routine just once and keep using the same lines (with exactly the same line numbers) again and again.

Here's how the PAUSE program looks without GOSUB\RETURN:

```
5 STALL = 1000
10 HOME
20 PRINT "A VERY IMPORTANT MESSAGE"
30 FOR PAUSE=1 TO STALL
35 NEXT PAUSE
40 HOME
50 PRINT "BE SURE TO SAVE YOUR PROGRAMS"
60 FOR PAUSE = 1 TO STALL
65 NEXT PAUSE
70 HOME
80 PRINT "BEFORE YOU TURN OFF YOUR COMPUTER!"
```

And here it is *with* GOSUB\RETURN. Type and run it:

```
NEW
5 STALL = 1000
10 HOME
20 MESSAGE$ = "A VERY IMPORTANT MESSAGE"
30 GOSUB 210 ———— Go to a subroutine at line 210.
40 MESSAGE$ = "BE SURE TO SAVE YOUR PROGRAMS"
50 GOSUB 210
60 MESSAGE$ = "BEFORE YOU TURN OFF YOUR COMPUTER!"
70 GOSUB 210
190 END ———— You must have this here.
200 REM **** MESSAGE SUBROUTINE *******
210 HOME ———— Subroutine starts here.
220 PRINT MESSAGE$
230 FOR PAUSE= 1 TO STALL
240 NEXT PAUSE
250 RETURN ———— Subroutine ends here; program returns to place
                that sent it here with GOSUB.
```

GOSUB means "Go to a subroutine." (A **subroutine** is a routine within a program reached through a GOSUB instruction.) Like GOTO, GOSUB makes the program go out of the normal sequence of line numbers to do something. Unlike GOTO, GOSUB returns to the point that it left; that's what RETURN does at the end of the subroutine. You don't have to keep track of the line number to go back to; GOSUB\RETURN keeps track for you.

END protects subroutines

Subroutines usually appear at the end of a program (*sub* is Latin for *under*), as in the examples in this session. You need to include an END instruction between your main program and your subroutines.

To see why, take out the END instruction at line 190 and run the program. (To take out a line, just type its line number and press Return.)

You got the error message RETURN WITHOUT GOSUB. Your computer expects to see a RETURN instruction only when a GOSUB sends it to a subroutine. If it encounters a RETURN by chance (as in this case), it doesn't know where to return to, gets confused, and tells you so with the error message.

One way to make sure your subroutines are isolated from the main program is to decide right away what line number your subroutines will start at, then put a line number and an END instruction right before that number. In the program you've been working with, the subroutine starts at line 210, just after the REM instruction at line 200; so the END instruction comes in line 190.

Subroutines and organization

In this next example, the code appears in subroutines, not because the program re-uses certain line segments often, but just because the program is easier to read and more organized that way. As you get better as a programmer, your programs tend to get longer and do more things. As that happens, having good organization in your code becomes more and more important.

Type and run this program. Note what's new about some of the lines that hold REM instructions:

```
NEW
5 REM  ***************************************
10 REM        Random Number Generator Program
12 REM        This program generates as many random numbers
14 REM        as the user wants. It also lets the user decide
16 REM        the range of numbers.
18 REM ***************************************
20 GOSUB 1010 : REM Title page
30 GOSUB 1110 : REM How many numbers & what range?
40 GOSUB 1210 : REM Generate random numbers
50 GOSUB 1310 : REM Go again?
60 IF ANS = "Y" THEN 30  : REM Repeat if yes...
70 PRINT
80 PRINT "Thanks for the screen space." : REM ...if not, end
999 END
1002 REM ******************************
1004 REM                              Title Page
1006 REM ******************************
1010 HOME
1020 PRINT "Random Number Generator"
```

```
1020 PRINT "Random Number Generator"
1030 PRINT
1040 PRINT "This program prints as many random numbers"
1050 PRINT "as you want between 0 and any limit you choose."
1060 PRINT
1070 PRINT
1080 INPUT "Press Return to start:    ";Start$
1090 RETURN
1102 REM ******************************
1104 REM             How many numbers & what limit?
1106 REM ******************************
1110 HOME
1120 INPUT "How many numbers do you want?  "; RNUMS
1130 PRINT
1140 INPUT "What's the highest a number can be?  "; LIMIT
1150 RETURN
1202 REM ******************************
1204 REM                          Generate Random Numbers
1206 REM ******************************
1210 FOR COUNT = 1 TO RNUMS
1220 NUM = RND( 1) * LIMIT
1230 PRINT NUM
1240 NEXT COUNT
1250 PRINT
1260 RETURN
1302 REM ***************
1304 REM                                  Go again?
1306 REM ***************
1310 INPUT "Do you want more random numbers? (Y/N)  "; ANS
1320 RETURN
```

This program uses a lot of subroutines to make it easier to see what's happening. Add to that all the REM instructions and the meaningful variable names, and you have a program that's especially easy to follow—both now, when you've just written it, and six months from now when you might decide to change a few of the lines.

Multiple instructions on one line

You've probably already figured out that you can have more than one instruction on a line if you put a colon (:) between instructions. Examples abound throughout the previous program. The program uses the colon only to add REM instructions, but you can use the colon with all instructions. Be careful, though; sometimes the results can surprise you.

For example, if you start a line with a REM instruction, your computer ignores the whole line and not just the REM instruction:

```
20 REM This whole line ignored : GOSUB 1010———GOSUB ignored!
```

We'll leave it to your own experimenting to discover other such surprises.

Organizing your programs: one step at a time

Sometimes the scope of a program feels overwhelming. It seems too complex or too long or just beyond your skill level. Sometimes that's true. You really don't have the ability to write a program that will control the nation's budget (and apparently, neither does anybody else). But you can do more than you probably realize with the things you've already learned. You can, for example, write a program to balance your checkbook.

The trick is to break down the task into easily manageable segments. Think for a moment how you balance your checkbook when you do it by hand:

1. Get the starting balance.
2. Add in the deposits.
   a. Get the amount of a deposit.
   b. Add that amount to the balance to produce a new balance.
   c. Keep doing steps a and b until all deposits are added in.
3. Subtract amounts for checks.
   a. Get the amount of a check.
   b. Subtract that amount from the balance to produce a new balance.
   c. Keep doing steps a and b until all checks are deducted.
4. Print the balance.

What you've just done is written out the **algorithm** (that is, the method to solve the problem) for balancing a checkbook. Your next step is to write modules for the steps in the algorithm; then all you need to do is line up the modules in the proper way. Program organization is a matter of lining up simple modules to work together.

The great checkbook balancing program challenge

Use the algorithm to write your own checkbook balancing program. Add a module that sets up a little menu so you can choose what to do first—add the total of checks written, or add up deposits.

After you've written your own version, have your computer print it out and then check it against the one listed here. Treat this as an opportunity to see how well you've understood what you've read in this tutorial. Take all the time you need; and remember to use REM lines liberally!

One version of a checkbook balancing program

This is just one version. If your version works, then it's just as good as this one. This version is here just in case you got stuck.

The important thing about this version is that it breaks the task down into simple steps:

Module 1

```
5 REM  ***********************
10 REM CHECKBOOK  BALANCER
15 REM ***********************
20 HOME
30 INPUT "Please type starting balance: $ "; BALANCE
40 PRINT
50 PRINT "1. Enter Deposits"
60 PRINT "2. Write Checks"
70 PRINT "3. End"
80 INPUT "CHOOSE BY NUMBER"; NUMBER
90 IF NUMBER = 1 THEN GOSUB 200
100 IF NUMBER = 2 THEN GOSUB 300
110 IF NUMBER = 3 THEN GOTO 17
120 IF NUMBER> 3 THEN GOTO 40          Traps out of range numbers.
130 PRINT "Your working balance is $ "; BALANCE
140 PRINT
150 INPUT "Press Return to continue: ";STALL$          Waits for user to be ready.
160 GOTO 40
170 PRINT
180 PRINT "Your ending balance is $ ";BALANCE
190 END
```

The first module represents the "body" of the program. Subroutines handle every other task. The next module handles deposits and adds them to the balance.

Module 2

```
200 REM ****************
210 REM MAKE DEPOSITS
220 REM ****************
230 HOME
240 INPUT "How many deposits did you make? "; ND
250 FOR X= 1 TO ND
260 INPUT "Amount of deposit: $ "; DEP
270 BALANCE = BALANCE + DEP          Keeps running total.
280 NEXT X
290 RETURN
```

Next, do the same thing for checks, except instead of adding to the balance, you subtract from it.

Module 3

```
300 REM ****************
310 REM WRITE CHECKS
320 REM ****************
330 HOME
340 INPUT "How many checks did you write? "; NC
350 FOR X= 1 TO NC
360 INPUT "Amount of check: $ "; CHECK
370 BALANCE = BALANCE - CHECK        Keeps running total.
380 NEXT X
390 RETURN
```

When you go over this program, it's easy to see what each part does. The REM lines show at a glance what happens in the subroutines.

Save your program onto the disk. In the next session, you'll learn how to make your programs more attractive, and this program will be a good one for you to practice on.

Summary and review

In this session, you learned about the GOSUB\RETURN instruction pair and about the importance of good program organization.

The GOSUB\RETURN instruction pair helps to organize programs into simple modules. Each subroutine is simply a task. Putting all the tasks together in an organized way is the secret to efficient programming. It's not how complex a program is, but rather how simple and well-organized it is. Keep that in mind, and you can tackle much larger tasks.

"Keep it simple" best summarizes this chapter. Break a program down into its component parts, and it becomes far easier to write.

Session 9

Formatting Screens

Generating information on a computer is exciting and rewarding. But the way you present the information is often just as important as the information itself. Just as a neatly organized and printed page conveys more information than a bunch of scribbles on a scrap of paper, so too does a well laid-out display have a greater impact than a barrage of characters hurled at the screen.

Clear screen presentation not only helps communicate ideas; it helps you organize your program as well. When you think about how something is going to look on your screen, you're also deciding what order your program must follow to get the results you want. Many programmers decide what all the screens are going to look like even before they begin to write the program.

This session teaches you the instructions and some of the techniques you need to create good screen presentations. You'll learn about placing text, highlighting important words, and creating menus. HTAB and VTAB let you place text anywhere on the screen. INVERSE lets you display uppercase text in dark characters against a light background (the opposite of what it usually is); NORMAL turns INVERSE off. You'll see how to control the placement of INPUT prompts. And you'll learn an algorithm for centering text.

Horizontal and vertical tabs

On a typewriter, you place your tab stops across the page. On your computer, you use HTAB to determine where the next tab stop will be.

Type and run this program:

```
NEW
10 HOME
20 HTAB 20
30 PRINT "HERE IT IS"
```

Without line 20, the message appears in the upper-left corner of your screen. The HTAB instruction makes the text begin twenty columns to the right. HTAB has a range from 0 to 255; you use it to place text anywhere across the screen. On the 40-column screen, each increment over 40 places the text one more line down. For example, HTAB 120 places text down three lines (120 / 40 = 3).

Type the following program and run it:

```
10 HOME
20 INPUT "HTAB value (0-255) "; HZ
30 HTAB HZ
40 PRINT "X"
50 PRINT
60 HTAB 20
70 INPUT "Another HTAB?(Y/N) "; ANS$
80 IF ANS$ = "Y" THEN 10
90 PRINT "Thanks for trying me out!"      Lets user know program's over.
100 END                                   Optional ending.
```

Usually you'll use HTAB just to position your text horizontally. To make vertical tabs, you'll use VTAB. VTAB works just like HTAB, but it can have values only from 1 to 24.

To get a quick idea of how VTAB works, run this next little program:

```
NEW
10 HOME
20 VTAB 10
30 PRINT "ABOUT HERE"
```

Combining HTAB and VTAB, you can place text anywhere on the screen.

This next program lets you experiment with putting things on your screen anywhere you want. Type and run it:

```
10 HOME
20 INPUT "HTAB position (1 - 40) "; HZ
25 IF HZ > 40 THEN PRINT "Too high!" : GOTO 20
30 INPUT "VTAB position (1 - 24) "; VT
35 IF VT > 24 THEN PRINT "Too high!" : GOTO 30
40 HOME
50 VTAB VT : HTAB HZ : PRINT "X"
60 VTAB 22 : HTAB 20
70 INPUT "Another? (Y/N) "; ANS$
80 IF ANS$ <> "N" THEN 10
90 PRINT "Bye, now."
100 END
```

In lines 50 and 60, the HTAB and VTAB instructions are on the same line. If you put HTAB and VTAB together like that, it's a little easier to organize text placement.

Making stylish program menus is easy with HTAB and VTAB. This next program uses a FOR\NEXT loop to generate positions for text.

```
NEW
10 HOME
20 FOR X=1 TO 6
30 HTAB 10 : VTAB (2 * X)
40 GOSUB 100
50 PRINT X; ".  "; MENU$
60 NEXT X
70 VTAB 20 : HTAB 5
80 INPUT "Choose by number:  "; NUMBER
90 END
100 REM ******************
110 REM MENU SELECTIONS
120 REM ******************
130 IF X = 1 THEN MENU$ = "Bring in the dog"
140 IF X = 2 THEN MENU$ = "Put out the cat"
150 IF X = 3 THEN MENU$ = "Feed the gorilla"
160 IF X = 4 THEN MENU$ = "Wash the seal"
170 IF X = 5 THEN MENU$ = "Pat the computer"
180 IF X = 6 THEN MENU$ = "END"
190 RETURN
```

Can you figure out what this does?

That menu doesn't do anything other than show you how to use HTAB and VTAB. But you can use this concept as a model in your own programs.

❖ *Why menus with numbers?* Good menus let users make choices by typing just one or two keystrokes. You can see how important good menu design can be when you look at this sample menu:

```
Which animal do you want information about?

Pachyderm
Pterodactyl
Ruffed Grouse
Serval
Programmorus Machinelinquae
Exit Program

Please type your choice here: ▮
```

This menu practically guarantees a typing error from all but the finest spellers. Your code will have to include all kinds of special error protection to check your user's typing. Numbered menus eliminate the problem:

```
Which animal do you want information about?

1)Pachyderm
2)Pterodactyl
3)Ruffed Grouse
4)Serval
5)Programmorus Machinelinquae
6)Exit Program

Please type your choice here (1 - 6): ▮
```

All your user has to do in this menu is type a number (and all your code has to check for is a numeric range). Numeric menus make things easier for both the user (who must type—and perhaps retype—choices) and for the programmer (who must write the code).

Prompt placement

Good screen design demands that you pay attention to how your INPUT prompts appear. Programmers often need to ask users for a number of inputs in a row—several street addresses, a number of prices, a series of names.

Type and run this program. It gets a series of inputs while keeping things neat. It uses HTAB and VTAB, plus a new programming trick:

```
NEW
10 GOSUB 200——————————————————You'll read about this later.
20 HOME
30 INPUT "How many names to enter? "; NAMES
40 HOME
50 HTAB 5 : VTAB 10
60 PRINT "Type in the names one at a time."
70 FOR X = 1 TO NAMES
80 HTAB 17 : VTAB 10 : PRINT SPACE$
90 HTAB 17 : VTAB 10
100 INPUT "Name:  "; NA$
110 NEXT X
120 END
200 REM ***************
210 REM SPACE MAKER
220 REM ***************
230 FOR S = 1 TO 20 : REM SPACE$ IS 20 SPACES LONG
240 SPACE$ = SPACE$ + " "
250 NEXT S
260 RETURN
```

Again, this is just a sample. In a "real" program, you wouldn't just get names and throw them away!

The SpaceMaker: The subroutine at line 200 introduces a nifty programming trick. You could have defined SPACE$ like this:

```
SPACE$ = "                    "——————20 spaces. Honest!
```

But that doesn't give you a very good idea of how many spaces are between the quotation marks. Using a loop to build SPACE$, as the subroutine at line 200 does, lets you see exactly how big the "blank out" space is going to be.

Of course, you aren't limited to just using spaces. Instead of using spaces, use dashes or underline characters. Be creative—just change what's between the quotation marks in line 240.

Getting noticed: INVERSE and NORMAL

Your computer can print inverse characters on the screen. The INVERSE instruction changes text from light-on-dark to dark-on-light. All text after an INVERSE instruction stays inverse until the program comes across a NORMAL instruction.

Type and run this little program for a quick demonstration:

```
NEW
10 HOME
20 INVERSE
30 PRINT  "THIS IS INVERSE"
40 NORMAL
50 PRINT "THIS IS NORMAL"
```

If you take out line 40, all of the text will be inverse. Because inverse text is more useful in getting attention than in presenting general displays, it's a good idea to put in the NORMAL instruction right after you've finished with INVERSE.

To get the user's attention when the program wants information, use an inverse prompt. For example, in a menu program, an inverse prompt separates it from the menu choices:

```
10 HOME
20 TITLE$ = "MENU"
30 PRINT TITLE$
40 FOR X = 1 TO 4
50 HTAB 3 : VTAB (2 * X) + 2            Figure this out yet?
60 PRINT  "Choice Number "; X
70 NEXT X
80 VTAB 20 : INVERSE                    Turns it on here...
90 INPUT "CHOOSE ONE: "; CHOOSE$
100 NORMAL                              ...and turns it off here.
110 ...
```

❖ *INVERSE IS FOR UPPERCASE ONLY:* INVERSE doesn't work well with lowercase letters. For perverse technical reasons, lowercase letters sometimes get changed to other characters when they're displayed in inverse. Experiment before you use lowercase letters in your programs, just to be sure.

Experiment some with INVERSE. Try making an inverse line of spaces. Put your name in inverse—in fact, use inverse text with asterisks to create a movie marquee and see your name in lights!

A text-centering algorithm

As you saw in the last session, algorithms are formulas written to perform different tasks. All of the tricks you've seen in these sessions are actually algorithms translated into computer code. As you've been experimenting with programs on your Apple, chances are you've developed some of your own algorithms. Most of the subroutines you've used are algorithms.

An algorithm for centering text is handy to have around, especially in a session on screen formatting. To construct that algorithm, you'll need to learn the LEN instruction. LEN calculates the length of a string. Here's an example:

```
NEW
10 A$="Apples Away!"
20 PRINT LEN( A$)
RUN
12
```

Parentheses are required (like RND).

Apples Away! has 12 characters (including the space).

When you center text, you put half the characters to the left of a line's midpoint and half the characters to the right.

Now that you have the basic idea, figure out on your own how the computer would see it. Write the code, try it out, and then read the solution in the next section.

One solution to the centering problem

Once again, this is just one possible solution. If yours is different and it works, then yours is just as valid as this one.

Here's the algorithm:

1. Get the number of characters that fit on one line (the screen width); that's either 40 or 80 on your Apple—choose the one you're using.
2. Find the string length by using LEN.

3. Subtract the string length from the screen width; divide the result by 2. The result is the position you're looking for.
4. Use HTAB to move to that position.

Expressed as computer code, it looks like this:

```
HTAB (WIDTH - LEN( LETTERS$) )/2
```
All these parentheses are necessary.

Use that algorithm in a program that will center any text you type in. Here's an example that keeps the algorithm in a subroutine. If your display is 80 columns, change 40 in line 130 to 80:

```
NEW
10 HOME
20 INPUT "Enter any word: "; W$
30 GOSUB 100
40 INVERSE : VTAB 20
50 INPUT "WOULD YOU LIKE ANOTHER (Y/N) "; ANS$
60 NORMAL : IF ANS$ = "Y" THEN 10
70 END
100 REM **************
110 REM CENTER TEXT
120 REM **************
130 HTAB (40 - LEN( W$))/2 : REM CENTERING ALGORITHM
140 VTAB 8
150 PRINT W$
160 RETURN
```

Summary and review

In this session, you learned about the importance of designing clear screen displays and about program menus.

Using HTAB and VTAB, you can place text anywhere on the screen. Text placement helps make clear what you're trying to say or what the program expects you to do next.

You learned that the INVERSE and NORMAL instructions separate and highlight text on your screen. These instructions help you make the program easier to use by highlighting important elements on the screen display.

You also learned that the soul of programming is algorithms. Like all other aspects of programming, you can build your own algorithms by reducing a task to a set of simple parts. Each algorithm, in turn, becomes a program building block.

Session 10

Programming for People

Congratulations! You've nearly completed your introduction to Applesoft BASIC and to the principles of programming. Most of the concepts you learned in this tutorial are traditional ones (as much as a science that's been around for only 45 years can have traditions). In this final session, you'll read about some even newer traditions, ones that have been developing only since the coming of personal computers. The ultimate goal is to get you to "humanize" your programs, to set them up in such a way that any computer novice can learn them quickly and use them easily.

You'll also read about how you can get a lot more help learning to program by joining a users group, taking programming classes, reading books, and subscribing to computer magazines.

A sordid history

Back in the old days (that is, before 1980 or so), programmers spent almost none of their time teaching their computers how to behave with humans. Programmers were primarily concerned with getting their programs to work without being stopped too often by error messages; because they themselves were usually the only people who used their programs, what they wrote didn't have to be "user-friendly." That was OK then; most people programmed for themselves and didn't share their programs with too many other people.

But remarkable changes have taken place over the last few years. Literally millions of people now own computers, and many thousands write programs for themselves, their business colleagues, and their friends. Most programmers, both hobbiests and professionals, belong to users groups—associations of computer owners who get together monthly (the fanatics do it weekly) to share their experiences, discoveries, and homemade programs.

If you're going to team up with this ever-growing group of sharing programmers (and if you continue to program, it's likely you will), it's important that you make your programs as easy to use as possible. The idea that programs and computers should be made for people and not the other way around is still revolutionary in a lot of circles. You are hereby officially invited to join the revolution.

People-program guidelines

Here are a few principles you can follow when you write programs for people. This list certainly doesn't exhaust the possible ways you can make your programs fit for human consumption, but it's enough to get you started:

Give Clear Prompts. To make it easy for your users to see what your program expects when it wants information, your program must communicate exactly what it wants. Prompts should stand out, be worded simply, and give the range of choices if there is a range.

Include Error Traps. People make mistakes. Your program should catch errors as much as it can and give your users a chance to make things right. Your program can easily check for the two most common problems: range errors and typing mistakes. In a range error, your user types in something that is beyond the range either of the computer or of the program:

```
90 ...
110 INPUT "Your choice - 0, 1, 2, or 3: "; CHOICE
120 IF CHOICE < 4 THEN GOTO 170                    Branches if choice OK.
130 PRINT "Sorry - Choice must be 0, 1, 2, or 3."  Error trap here.
140 PRINT "Please make another choice."
150 PRINT
160 GOTO 110                                       Goes back for another try.
170 ...                                            Comes here if OK.
```

In a typing mistake, your user types something he or she didn't mean, or makes a simple spelling error:

```
90 ...
100 INPUT "Name of program to erase: "; ERASE$
110 PRINT
120 INVERSE
130 PRINT "WARNING!  IF YOU ERASE"                 Gives a warning.
140 PRINT ERASE$                                   Reprints entry.
150 PRINT "IT'S GONE FOREVER!  "
160 PRINT
170 NORMAL
180 INPUT "WIPE OUT THE PROGRAM? (Y/N) "; KILL$
190 IF KILL$ <> "Y" THEN HOME:  GOTO 100           Cancels if not verified.
200 ...
```

Leave an Exit Open. Don't forget to give users a way out of your program. As wonderful as your program might be to use, people do like to do other things like eat, go to school, and take vacations. There are several ways you can determine when your user has finished using your program.

For example, you can have a question at the outset to ask how many entries the user needs to make:

```
90 ...
100 INPUT "How many checks did you write? "; CHECKS
110 FOR X = 1 TO CHECKS
120 ...
```

Or you can give an exit option after each entry:

```
90 ...
100 INPUT "How much is the next check for? $ "; AMOUNT
110 BALANCE = BALANCE - AMOUNT
120 INPUT "Another check? (Y/N): ";ANS$
130 IF ANS$ = "N" THEN GOSUB 1000 : REM Show balance and end
140 ...
```

Or you can have the program return to a menu with an exit option after each entry or series of entries:

```
90 ...
100 PRINT "1.  Enter more names"
110 PRINT "2.  Change an entry"
120 PRINT "3.  Print out all entries"
130 PRINT "4.  Leave the program"
140 VTAB 22 : HTAB 25
150 INPUT "Your choice: "; CHOICE
160 IF CHOICE = 4 THEN END
170 ...
```

To see the rules in action, type and run each of the following two programs; the first doesn't follow the rules and the second does:

Nerd Programming (Yuch)

```
10 INPUT A
20 SUM=SUM + A
30 PRINT SUM
40 GOTO 10
```

People Programming (Fantastic)

```
10 HOME
20 INPUT "Amount to add (0 to stop)"; AMOUNT : REM Get amount.
30 IF AMOUNT = 0 THEN GOTO 130 : REM End if user's through.
40 PRINT
50 PRINT "You added"; AMOUNT; ", right? (Y/N)";
60 INPUT ""; YN$ : REM Entry OK?
70 IF YN$ ="N" THEN GOTO 20 : REM If not, get it again.
80 SUM = SUM + AMOUNT : REM Keep Running total...
90 PRINT
100 PRINT "Your running total is "; SUM: REM ...and report it.
110 PRINT
120 GOTO 20 : REM     Get another number
130 PRINT
140 PRINT "Final total: "; SUM : REM Print the final total.
```

Humanizing programs isn't easy

The second program requires more work than the first one. It takes more planning, more typing, and more debugging to write a good **interactive program** (that is, one that talks to people). It is also worth it. Real people make mistakes; write programs with that in mind.

It gets easier

The more you learn about programming, the easier it gets. After you've been programming for a while, you'll find that what once took you twenty lines of programming you may do in only five lines. By experimenting, playing, and trying new things with your Apple computer, your programming ability will grow quicker than you can imagine.

Where do you go from here?

If you decide that programming's not for you, then there's no problem. You don't have to know how an internal combustion engine works to drive a car, and you don't have to know how to program to use a computer. But if you've enjoyed going through this tutorial and you've decided that programming is fun and interesting, you can do lots of things to help yourself learn more.

Read Books on Applesoft Programming: Hundreds of books have been written on Applesoft, from tutorials to advanced technical documents. Any decent bookstore has at least a few Applesoft titles; the larger stores carry dozens. The absolutely indispensible resource is the *Applesoft BASIC Programmer's Reference Manual*, published by Addison-Wesley (ISBN 0-201-17722-6). Written by the experts at Apple Computer, Inc., this is *the* official Applesoft book. Your Apple Computer dealer or local bookstore carries it or can order it for you.

Join an Apple Users Group: Made up of people at all levels of expertise, Apple users groups are a new computerist's best friend. As each member learns something, he or she passes it on to the others. Most clubs have special subgroups for beginners; virtually all of them have special interest subgroups for learning Applesoft BASIC, as well as for other computer languages. (Logo, Pascal, C, and Forth are the most popular ones.) Besides being practical, these groups are a lot of fun.

❖ *Free software!* One of the best ways to learn how to write programs is to look at somebody else's. When you join an Apple users group, you'll have access to tons of public domain software. And many public domain programs are written in Applesoft.

Programming Classes: You can find programming classes in high schools, universities, community colleges, computer stores, specialty schools, and users groups. Check with the instructor about the level of the class before you take it; if possible, talk to some graduates. Then you'll be sure that the instruction is at the level you want.

Subscribe to Magazines About Apple Computers: There are dozens of computer magazines, many specializing in Apple computers. See if you can find one that deals exclusively with your model of Apple. Some Apple magazines cover both Macintosh and Apple II family computers, while others cover only one or the other. And some are aimed more at program users than at program writers. Again, this is an area where a users group can really help out. Not only can members recommend magazines that have beginners' columns, but many clubs have libraries of back issues you can use.

Do it!

The most important thing you can do to learn to program is—to program. Write silly programs and serious programs, long and short programs, programs that are fancy, and programs that are plain. Just do it! You'll learn more from an hour of mistakes than from a week's listening in a classroom. Code to your heart's content.

A parting word

This brief book has been a guided exploration through some of the most important concepts in elementary programming. You didn't learn all of the instructions in Applesoft BASIC; there are far too many of them to teach in one short manual. But what you learned here can serve you well if, whenever you write a program, you remember that you're writing for other people.

And keep on coding!

Appendix A

A Summary of Applesoft Instructions

This is a brief summary of all the instructions in the Applesoft BASIC language. This summary is included for those programmers already proficient in some other computer language, but new to Applesoft BASIC.

For a complete description of these instructions, see the *Applesoft BASIC Programmer's Reference Manual* (Addison-Wesley Publishing Company, Inc.).

ABS

```
ABS (-2.77)
```

Yields the absolute value (value without regard to sign) of the argument. The example yields 2.77.

ASC

```
ASC ("QUEST")
```

Yields the ASCII code for the first character in the argument. The example yields 81 (ASCII code for Q).

Assignment Instruction

```
LET A = 23.567
A$ = "HUMBUG"
```

Assigns the value of the expression following = to the variable preceding it. LET is optional.

ATN

```
ATN (.8771)
```

Yields the arc tangent, in radians, of the argument. The example yields .720001187 (radians).

CALL

```
CALL -922
```

Executes a machine-language subroutine at the specified decimal memory address. The example issues a line feed.

CHR$

```
CHR$ (65)
```

Yields the character corresponding to the ASCII code given as an argument. The example yields the letter A.

CLEAR

```
CLEAR
```

Resets all variables and internal control information to their initial state. Program code is unaffected.

COLOR=

```
COLOR= 12
```

Sets the display color for plotting low-resolution graphics. The example sets the display color to green.

CONT

```
CONT
```

Resumes program execution after it has been halted by STOP, END, CONTROL-C, or (sometimes) CONTROL-RESET.

COS

```
COS (2)
```

Yields the cosine of the argument, which must be expressed in radians. The example yields -.416146836.

DATA

```
DATA JOHN SMITH, "CODE 32", 23.45, -6
```

Creates a list of items for use by READ instructions. In the example, the first item is the string JOHN SMITH, the second is the string "CODE 32", the third is the real number 23.45, and the fourth is the integer -6.

DEF FN

```
DEF FN CUBE (X) = X * X * X
```

Defines a new function for use in the program. The example defines a function that yields the cube of its argument.

DEL

```
DEL 23, 56
```

Deletes a range of consecutive lines from the program. The example deletes lines 23 to 56, inclusive.

DIM

```
DIM MARK (50,3), NAMES (50)
```

Defines and allocates space for one or more arrays. The example defines a two-dimensional real array MARK, whose first subscript varies from 0 to 50 and whose second varies from 0 to 3, and a string array NAME$ with one subscript that varies from 0 to 50.

DRAW

```
DRAW 4 AT 50,100
DRAW 4
```

Draws a shape at a specified point on the high-resolution graphics screen from the shape table currently in memory. The first example draws shape number 4, beginning in column 50, row 100, using the current color, scale, and rotation settings; the second example draws shape 4 at the last point plotted by HPLOT, DRAW, or XDRAW.

END

```
END
```

Terminates the execution of the program and returns control to the user. No message is displayed.

EXP

```
EXP (2)
```

Yields the mathematical exponential of its argument (that is, the constant e—2.7182818—raised to the power specified by the argument). The example yields e squared, or 7.3890561.

FLASH

```
FLASH
```

Causes all text displayed on the screen with subsequent PRINT statements to flash between light-on-dark and dark-on-light. May not work properly for lowercase letters (and other characters with ASCII codes above 95) if the computer is running in "active-80" mode.

FN

```
FN CUBE (6)
```

Applies a designated function to the value of the argument expression. Assuming the definition for the function CUBE given under DEF FN, the example yields the value 216.

FOR

```
FOR J = 1 TO 10
FOR MARK= 0 TO 100 STEP 5
FOR NUMBER = 20 TO -20 STEP -2
```

Marks the beginning of a loop, identifies the index variable, and gives the variable's starting and ending values and (optionally) the amount by which it is to change (step) on each pass through the loop. The first example begins a loop whose index variable J takes on all values from 1 to 10, stepping by 1; the second begins a loop whose index variable MARK takes on values from 0 to 100, stepping by 5; the third begins a loop whose index variable NUMBER takes on values from 20 to -20, stepping by -2.

FRE

```
FRE (0)
```

Yields the amount of remaining memory, in bytes, available to the program. Also forces "garbage collection" of dead strings. The argument is ignored, but must be a valid Applesoft expression.

GET

```
GET ANSWER$
```

Accepts a single character from the keyboard without displaying it on the screen and without requiring that the Return key be pressed. Program execution is suspended until the user presses a key. In the example, the character typed is assigned to the variable ANSWER$.

GOSUB

```
GOSUB 250
```

Executes a subroutine beginning at the designated line number (250 in the example).

GOTO

```
GOTO 400
```

Sends control unconditionally to the designated line number (400 in the example).

GR

```
GR
```

Converts the display to 40 rows of low-resolution graphics with four lines of text at the bottom. The screen is cleared to dark, the cursor is moved to the beginning of the last line, and the low-resolution display color is set to black.

HCOLOR=

```
HCOLOR= 1
```

Sets the display color for plotting high-resolution graphics. The example sets the display color to green.

HGR

```
HGR
```

Converts the display to 160 rows of high-resolution graphics with four lines for text at the bottom. The screen is cleared to black and page 1 of high-resolution graphics is displayed. The contents of the text display, the location of the cursor, and the high-resolution display color are unaffected.

HGR2

```
HGR2
```

Converts the display to full-screen (192 rows) high-resolution graphics with no text. The screen is cleared to black and page 2 of high-resolution graphics is displayed. The contents of the text display, the location of the cursor, and the high-resolution display color are unaffected.

HIMEM:

```
HIMEM: 32767
```

Sets the address of the highest memory location available to the Applesoft program, including its variables. The example sets the end of program and variable storage to 32767. Used to protect an area of memory for data, high-resolution graphics, or machine-language code.

HLIN

```
HLIN 10, 20 AT 30
```

Draws a horizontal line in low-resolution graphics, using the current low-resolution display color. The example draws a line across row 30 from column 10 to column 20.

HOME

```
HOME
```

Clears all text from the text window and moves the cursor to the top-left corner of the window.

HPLOT

```
HPLOT 75, 20
HPLOT 48, 115 TO 79, 84 TO 110, 115
HPLOT TO 270, 10
```

Plots a point or line on the high-resolution graphics screen in the current high-resolution display color. The first example plots a single point at column 75, row 20; the second example draws lines from column 48, row 115 to column 79, row 84 to column 110, row 115; the third draws a line to column 270, row 10 from the last point plotted with HPLOT, using the color of the last point plotted (not necessarily the current display color).

HTAB

```
HTAB 23
```

Positions the cursor to a specified column of the text display. The example moves the cursor to column 23.

IF...THEN

```
IF AGE< 18 THEN A = 0 : B = 1: C = 2
IF ANSWER$ = "YES" THEN GOTO 100
IF N > MAX THEN GOTO 25
IF N > MAX THEN 25
IF N > MAX GOTO 25
```

Executes or skips one or more instructions, depending on the truth of a stated condition. The first example sets A to 0, B to 1, and C to 2 if the value of AGE is less than 18; the second branches to line 100 if the value of ANSWER$ is the string "YES"; the last three all branch to line 25 if the value of N is greater than that of MAX. In all cases, if the stated condition is false, execution continues with the next program line.

IN#

```
IN# 2
```

Specifies the source for subsequent input. The example causes subsequent input to be read from the device at port 2.

INPUT

```
INPUT A%
INPUT "TYPE AGE, THEN A COMMA, THEN NAME "; AGE, NAME$
```

Reads a line of input from the current input device. The first example reads a value into variable A%; the second displays a prompting message and then reads values into variables AGE and NAME$.

INT

```
INT (98.6)
INT (-273. 16)
```

Yields the integer part of the argument value. The examples yield 98 and -274, respectively.

INVERSE

```
INVERSE
```

Causes all uppercase text displayed on the screen with subsequent PRINT instructions to appear in dark-on-light instead of the usual light-on-dark. Has unpredictable effects on lowercase text.

LEFT$

```
LEFT$ ("APPLESOFT", 5)
```

Yields a specified number of characters from the beginning of a string. The example yields the string APPLE.

LEN

```
LEN ("NEVER A DULL MOMENT")
```

Yields the length of a string in characters. The example yields 19.

LET

See "Assignment Instruction."

LIST

```
LIST
LIST 150
LIST 200-300
LIST 200, 300
```

Displays all or part of the program on the screen, or writes it to the current output device. The first example lists the entire program; the second lists line 150 only; the last two list lines 200 to 300, inclusive.

LOAD

```
LOAD DEMO
```

Reads a program into memory from a disk. The example reads a program from a disk file named DEMO.

LOG

```
LOG (2)
```

Yields the natural logarithm of the argument. The example yields .693147181.

LOMEM:

```
LOMEM: 24576
```

Sets the address of the lowest memory location available to the program for variable storage. The example sets the beginning of variable storage to 24576.

MID$

```
MID$ ("AN APPLE A DAY", 4, 5)
MID$ ("AN APPLE A DAY", 4)
```

Yields a specified number of characters beginning at a specified position in a given string. The first example yields the string `APPLE`; the second yields the string `APPLE A DAY`.

NEW

```
NEW
```

Clears the current program from memory and resets all variables and internal control information to their initial states.

NEXT

```
NEXT
NEXT INDEX
NEXT J, I
```

Marks the end of a loop and causes the loop to be repeated for the next value of the index variable, as specified in the corresponding FOR instruction. The first example ends the most recently entered loop; the second ends the loop whose index variable is INDEX; the third ends the pair of nested loops whose index variables are J and I.

NORMAL

```
NORMAL
```

Causes all text displayed on the screen with subsequent PRINT instructions to appear in the usual light-on-dark; cancels the effects of INVERSE.

NOTRACE

```
NOTRACE
```

Stops the display of line numbers for each instruction executed; cancels the effects of TRACE.

ON ... GOSUB

```
ON ID GOSUB 100, 200, 23, 4005, 500
```

Chooses a subroutine to execute depending on the value of an expression. The example transfers control to the subroutine beginning at line 100, 200, 23, 4005, or 500, depending on whether the value of ID is 1, 2, 3, 4, or 5; if ID has none of these values, execution continues with the next instruction.

ON ... GOTO

```
ON ID GOTO 100, 200, 23, 4005, 500
```

Chooses a line number to branch to depending on the value of an expression. The example transfers control to line 100, 200, 23, 4005, or 500, depending on whether the value of ID is 1, 2, 3, 4, or 5; if ID has none of these values, execution continues with the next instruction.

ONERR GOTO

```
ONERR GOTO 500
```

Replaces Applesoft's normal error-handling mechanism with a subroutine beginning at a specified line number. The example establishes an error-handling subroutine beginning at line 500.

PDL

```
PDL (1)
```

Reads the current dial setting on a designated hand control. The example reads the dial on hand control 1.

PEEK

```
PEEK (37)
```

Yields the contents of a specified location in memory. The example yields the contents of location 37, which contains the current vertical position of the text cursor on the display screen.

PLOT

```
PLOT 10, 20
```

Plots a single block of the current display color at a specified position on the low-resolution graphics screen. The example plots a block at column 10, row 20.

POKE

```
POKE -16302, 0
```

Stores a value in a specified location in memory. The example stores the value 0 at location 49234 (65536 - 16302), causing the display to switch from mixed graphics and text to full-screen graphics.

POP

```
POP
```

Removes the most recent return address from the control stack, causing the next RETURN instruction to send control to the instruction following the second most recently executed GOSUB.

POS

```
POS (0)
```

Yields the current horizontal position of the cursor on the text display. The argument is ignored, but must be a valid Applesoft expression.

PR#

```
PR# 1
```

Specifies the destination for subsequent output. The example causes subsequent output to be sent to the device at port 1.

PRINT

```
PRINT
PRINT A$, "X = "; X
```

Writes a line of output to the current output device. The first example writes a blank line; the second writes the value of variable A$, followed at the next available tab position by the string "X = ", followed immediately by the value of variable X.

READ

`READ A, B%, C$`

Reads values from DATA instructions in the body of the program. The example reads values into variables A, B%, and C$.

REM

`REM THIS A REMARK`

Includes remarks in the body of a program for the benefit of a human reader.

RESTORE

`RESTORE`

Causes the next READ instruction executed to begin reading at the first item of the first DATA instruction in the program.

RESUME

`RESUME`

At the end of an error-handling routine (see ONERR GOTO), causes resumption of the program at the beginning of the instruction in which the error occurred.

RETURN

`RETURN`

The last instruction in a subroutine returns control from a subroutine to the instruction following the GOSUB that called the subroutine.

RIGHT$

`RIGHT$ ("APPLESOFT", 4)`

Yields a specified number of characters from the end of a string. The example yields the string `SOFT`.

RND

`RND (1)`

Yields a random number between 0 and 1. Zero and negative argument values yield repeatable sequences of random numbers.

ROT=

`ROT= 16`

Sets the angular rotation for high-resolution shapes to be drawn with DRAW or XDRAW. The example causes the shape to be rotated 90 degrees clockwise.

RUN

```
RUN
RUN 500
RUN DEMO
```

Executes an Applesoft program. The first example executes the program currently in memory from the beginning; the second executes the program in memory, starting at line 500; the third loads and executes a program from a disk file named DEMO.

SAVE

```
SAVE DEMO
```

Writes the named Applesoft program currently in memory to a disk. The example writes the program to a disk file named DEMO.

SCALE=

```
SCALE= 10
```

Sets the scale factor for high-resolution shapes to be drawn with DRAW or XDRAW. The example causes the shape to be drawn ten times bigger than the definition given in the shape table.

SCRN

```
SCRN (10, 20)
```

Yields the code for the color currently displayed at a designated position on the low-resolution graphics screen. The example yields the code for the color at column 10, row 20.

SGN

```
SGN (-144)
```

Yields a value of -1, 0, or +1, depending on the sign of the argument. The example yields -1.

SIN

```
SIN (2)
```

Yields the sine of the argument, which must be expressed in radians. The example yields .909297427.

SPC

```
SPC (8)
```

Introduces a specified number of spaces into the line being written by a PRINT instruction. The example writes eight spaces.

SPEED=

```
SPEED= 50
```

Sets the rate at which text characters are to be sent to the display screen or other input/output device. The slowest rate is 0; the fastest is 255.

SQR

```
SQR(2)
```

Yields the positive square root of the argument; the example yields 1.41421356.

STOP

```
STOP
```

Terminates the execution of the program and returns control to the user. A message is displayed identifying the program line in which the STOP instruction appears.

STR$

```
STR$ (12.45)
```

Yields a string representing the numeric value of the argument. The example yields the string "12.45".

TAB

```
TAB (23)
```

Positions the text cursor at a specified position on the output line during execution of a PRINT instruction. The example moves the cursor to column 23.

TAN

```
TAN (2)
```

Yields the tangent of the argument, which must be expressed in radians. The example yields -2.18503987.

TEXT

```
TEXT
```

Converts the display to 24 lines of text, with the cursor positioned at the beginning of the bottom line.

TRACE

```
TRACE
```

Causes the line number of each instruction to be displayed on the screen as it is executed.

USR

```
USR (3)
```

Executes a machine-language subroutine supplied by the user, passing it a specified argument. The subroutine is entered via a JMP (jump) instruction stored at addresses $0A through $0C hexadecimal. The example passes the argument value 3.

VAL

```
VAL ("-3.7E4")
```

Yields the numeric value represented by the string supplied as an argument. The example yields -37000.

VLIN

```
VLIN 10, 20 AT 30
```

Draws a vertical line in low-resolution graphics, using the current low-resolution display color. The example draws a line down column 30 from row 10 to row 20.

VTAB

```
VTAB 15
```

Positions the cursor to a specified row of the text display. The example moves the cursor to row 15

WAIT

```
WAIT 49347, 15
WAIT 49347, 15, 12
```

Suspends program execution until a specified bit pattern appears at a specified memory location. Used to wait for a status signal from a peripheral device. The second and (optional) third arguments are masks: the second specifies which bits of the designated location are of interest, the third specifies the values to be tested for in those bits. The first example suspends execution until a 1 bit appears in any of the four low-order bit positions of location 49347; the second waits for a 1 bit in position 0 or 1 or a 0 bit in position 2 or 3.

XDRAW

```
XDRAW 4 AT 50, 100
XDRAW 4
```

Draws a shape from the shape table currently in memory at a specified point on the high-resolution graphics screen. Each point in the shape is plotted using the complement of the color currently displayed at that point. Typically used to erase a shape already drawn. The first example erases shape number 4, beginning in column 50, row 100, using the current scale and rotation settings; the second example erases shape 4 at the last point plotted by HPLOT, DRAW, or XDRAW.

Appendix B

Reserved Words

Table B-1 shows a list of Applesoft's reserved words. In most cases these character sequences cannot be used as, or embedded in, variable names.

The ampersand character (&) is reserved for Applesoft's internal use and for user-supplied machine-language routines.

XPLOT is a reserved word that does not correspond to a current Applesoft statement.

Some reserved words are recognized by Applesoft only in certain contexts:

COLOR, HCOLOR, ROT, SCALE, and SPEED are interpreted as reserved words only if the next nonspace character is an equal sign (=). This is of little benefit in the case of COLOR and HCOLOR, as the embedded reserved word OR prevents their use as variable names anyway.

HIMEM and LOMEM are interpreted as reserved words only if the next nonspace character is a colon (:).

IN and PR are interpreted as reserved words only if the next nonspace character is a number sign (#).

SCRN, SPC, and TAB are interpreted as reserved words only if the next nonspace character is a left parenthesis, (.

ATN is interpreted as a reserved word only if there is no space between the T and the N. If a space occurs between the T and the N, the reserved word AT is interpreted instead of ATN.

TO is interpreted as a reserved word unless it is preceded by an A and there is a space between the T and the O. In that case, the reserved word AT is interpreted instead of TO.

Even if you don't embed reserved words in your variable names, they can sometimes pop up unexpectedly and cause problems. For example, the statement

```
100 FOR A = LOFT OR LEFT TO 15
```

is interpreted as

```
100 FOR A = LOF TO RLEFT TO 15
```

and causes a syntax error. To force the correct interpretation, use parentheses:

```
100 FOR A = (LOFT) OR (LEFT) TO 15
```

Table B-1 Applesoft Reserved Words

$	FLASH	IF	ON	SAVE	USR
	FN	IN#	PDL	SCALE=	
ATN			PEEK	SCRN(	
	GET	LEFT$	PLOT	SGN	VAL
CALL	GOSUB	LEN	POKE	SHLOAD	VLIN
CHR$	GOTO	LET	POP	SIN	VTAB
CLEAR	GR	LIST	POS	SPC(	
COLOR=		LOAD	PRINT	SPEED=	
CONT		LOG	PR#	SQR	WAIT
COS	HCOLOR=	LOMEM		STEP	
	HGR			STOP	
DATA	HGR2		READ	STORE	XPLOT
DEF	HIMEM:	MID$	RECALL	STR$	XDRAW
DEL	HLIN		RESTORE		
DIM	HOME		RESUME		
DRAW	HPLOT	NEW	RETURN	TAB(	
	HTAB	NEXT	RIGHT$	TAN	
END		NORMAL	RND	TEXT	
EXP		NOT	ROT=	THEN	
		NOTRACE	RUN	TO	
				TRACE	

Glossary

address: A number used to identify something, such as a location in the computer's memory.

algorithm: A step-by-step procedure for solving a problem or accomplishing a task.

Apple II: A family of personal computers, manufactured and sold by Apple Computer, Inc.; generic name for all computers in the series.

Applesoft: An extended version of the BASIC programming language used with the Apple II family of computers and capable of processing numbers in floating-point form. An interpreter for creating and executing programs in Applesoft is built into the Apple II system in ROM.

arithmetic operator: An operator, such as +, that combines numeric values to produce a numeric result; compare **relational operator.**

BASIC: *Beginners All-purpose Symbolic Instruction Code*; a high-level programming language designed to be easy to learn and use.

branch: To send program execution to a line or instruction other than the next in sequence.

bug: An error in a program that causes it not to work as intended.

catalog: A list of all files stored on a disk; sometimes called a **directory**.

character: A letter, digit, punctuation mark, or other written symbol used in printing or displaying information in a form readable by humans.

code: (1) A number or symbol used to represent some piece of information in a compact or easily processed form. (2) The statements or instructions making up a program.

command: A communication from the user to a computer system (usually typed from the keyboard) directing it to perform some immediate action.

computer: An electronic device for performing predefined (programmed) computations at high speed and with great accuracy.

computer system: A computer and its associated hardware, firmware, and software.

concatenate: Literally, "to chain together"; to combine two or more strings into a single, longer string containing all the characters in the original strings.

conditional branch: A branch that depends on the truth of a condition or the value of an expression.

control variable: see **index variable**.

counter: A variable used to keep track of passes through a loop. Counters often have the form X = X + 1.

crash: When a program unexpectedly ceases operating, possibly damaging or destroying information in the process.

cursor: A marker or symbol displayed on the screen that marks where the user's next action will take effect or where the next character typed from the keyboard will appear.

debug: To locate and correct an error or the cause of a problem or malfunction in a computer system. Typically used to refer to software-related problems.

deferred execution: The saving of an Applesoft program line for execution at a later time as part of a complete program; occurs when the line is typed with a line number. Compare **immediate execution**.

delay loop: A loop whose purpose is to slow down the execution of a program.

define: To assign a value to a variable.

disk: An information-storage medium consisting of a flat, circular magnetic surface on which information can be recorded in the form of small magnetized spots, similarly to the way sounds are recorded on tape.

disk drive: A peripheral device that writes and reads information on the surface of a magnetic disk.

display: (1) Information exhibited visually, especially on the screen of a video display device. (2) To exhibit information visually. (3) A display device.

display device: A device that exhibits information visually, such as a television receiver or video monitor.

display screen: The glass or plastic panel on the front of a display device, on which images are displayed.

edit: To change or modify; for example, to insert, remove, replace, or move text in a document.

error message: A message displayed or printed to notify the user of an error or problem in the execution of a program.

execute: To perform or carry out a specified action or sequence of actions, such as those defined by a program.

file: A collection of information stored as a named unit on a peripheral storage medium such as a disk.

filename: The name under which a file is stored on a disk.

firmware: Name applied to programs stored in **read-only memory**.

format: (1) The form in which information is organized or presented. (2) To specify or control the format of information. (3) To prepare a blank disk to receive information by dividing its surface into tracks and sectors; also **initialize**.

graphics: (1) Information presented in the form of pictures or images. (2) The display of pictures or images on a computer's display screen. Compare **text**.

hacker: An experienced programmer.

hand control: An optional peripheral device that can be connected to the Apple II's hand control connector and has a rotating dial and a push button; typically used to control game-playing programs, but can be used in more serious applications as well.

hang: For a program or system to "spin its wheels" indefinitely, performing no useful work.

hard copy: Information printed on paper for human use.

immediate execution: The execution of an Applesoft program line as soon as it is typed; occurs when the line is typed without a line number. Compare **deferred execution**.

index variable: A variable whose value changes on each pass through a loop; often called **control variable** or **loop variable**.

infinite loop: A section of a program that repeats the same sequence of actions indefinitely.

information: Facts, concepts, or instructions represented in an organized form.

initialize: (1) To set to an initial state or value in preparation for some computation. (2) To prepare a blank disk to receive information by dividing its surface into tracks and sectors; also **format**.

input: (1) Information transferred into a computer from some external source, such as the keyboard, a disk drive, or a modem. (2) The act or process of transferring such information.

input variable: Variable whose value is assigned by the user via an INPUT instruction, as opposed to one whose value is assigned by the programmer using an assignment or similar instruction.

instruction: A unit of a program in a high-level programming language that specifies an action for the computer to perform, typically corresponding to several instructions of machine language.

interactive: Operating by means of a dialog between the computer system and a human user.

interactive programming: Generating programs that operate by means of a dialog between the computer system and a human user.

interface: The devices, rules, or conventions by which one component of a system communicates with another.

inverse video: The display of text on the computer's display screen in the form of dark dots on a light (or other single phosphor color) background, instead of the usual light dots on a dark background.

keyboard: The set of keys, similar to a typewriter keyboard, for typing information to the computer.

language: See **programming language**.

line: See **program line**.

line number: A number that identifies a program line in an Applesoft program.

load: To transfer information from a peripheral storage medium (such as a disk) into main memory for use; for example, to transfer a program into memory for execution.

loop: A section of a program that is executed repeatedly until some condition is met, such as an index variable reaching a specified ending value.

loop variable: See **index variable**.

low-resolution graphics: The display of graphics on the Apple II's display screen as a sixteen-color array of blocks, 40 columns wide and either 40 or 48 rows high.

memory: A component of a computer system that can store information for later retrieval; see **main memory**, **random-access memory**, **read-only memory**.

menu: A list of choices presented by a program, usually on the display screen, from which the user can select.

mode: (1) Any of several ways a computer interprets information. (2) A state of a computer or system that determines its behavior.

nested loop: A loop contained within the body of another loop and executed repeatedly during each pass through the containing loop.

nested subroutine call: A call to a subroutine from within the body of another subroutine.

numeric variable: see **variable**.

operator: A symbol or sequence of characters, such as + or AND, specifying an operation to be performed on one or more values (the operands) to produce a result.

output: (1) Information transferred from a computer to some external destination, such as the display screen, a disk drive, a printer, or a modem. (2) The act or process of transferring such information.

pass: A single execution of a loop.

precedence: The order in which operators are applied in evaluating an expression.

printer: A peripheral device that writes information on paper in a form easily readable by humans.

program: (1) A set of instructions that describes actions for a computer to perform in order to accomplish some task, conforming to the rules and conventions of a particular programming language. In Applesoft, a sequence of program lines, each with a different line number. (2) To write a program.

program line: The basic unit of an Applesoft program, consisting of one or more instructions separated by colons (:).

programmer: The human author of a program; one who writes programs.

programming: The activity of writing programs.

programming language: A set of rules or conventions for writing programs.

prompt: (1) To remind or signal the user that some action is expected, typically by displaying a distinctive symbol, a reminder message, or a menu of choices on the display screen. (2) An instruction or reminder message that appears on the display screen.

prompt character: (1) A text character displayed on the screen to prompt the user for some action. Often also identifies the program or component of the system that is doing the prompting; for example, the prompt character] is used by the Applesoft BASIC Interpreter. Also called *prompting character.* (2) Someone who is always on time.

prompt message: A message displayed on the screen to prompt the user for some action. Also called *prompting message.*

RAM: See **random-access memory**.

random-access memory: Memory whose contents can be both read and written; often called *read-write memory.* The contents of an individual location in random-access memory can be referred to in an arbitrary or random order. The information contained in this type of memory is erased when the computer's power is turned off, and is permanently lost unless it has been saved on a more permanent storage medium, such as a disk. Compare **read-only memory**.

read: To transfer information into the computer's memory from a source external to the computer (such as a disk drive or modem) or into the computer's processor from a source external to the processor (such as the keyboard or main memory).

read-only memory: Memory whose contents can be read but not written; used for storing **firmware**. Information is written into read-only memory once, during manufacture; it then remains there permanently, even when the computer's power is turned off, and can never be erased or changed. Compare **random-access memory**.

read-write memory: See **random-access memory**.

relational operator: An operator, such as >, that compares numeric values to produce a logical result; compare **arithmetic operator**.

reserved word: A word or sequence of characters reserved by a programming language for some special use, and therefore unavailable as a variable name in a program.

ROM: See **read-only memory**.

routine: A part of a program that accomplishes some task subordinate to the overall task of the program.

run: (1) To execute a program. (2) To load a program into main memory from a peripheral storage medium, such as a disk, and execute it.

save: To transfer information from main memory to a peripheral storage medium for later use.

screen: See **display screen**.

starting value: The value assigned to the index variable on the first pass through a loop.

step value: The amount by which the index variable changes on each pass through a loop.

stepwise refinement: A technique of program development in which broad sections of the program are laid out first, then elaborated step by step until a complete program is obtained.

string: An item of information consisting of a sequence of text characters.

string variable: see **variable**.

subroutine: A part of a program that can be executed on request from any point in the program, and that returns control to the point of the request on completion.

syntax: The rules governing the structure of statements or instructions in a programming language.

system: A coordinated collection of interrelated and interacting parts organized to perform some function or achieve some purpose.

text: (1) Information presented in the form of characters readable by humans. (2) The display of characters on the Apple II's display screen. Compare **graphics**.

user: The person operating or controlling a computer system.

user interface: The rules and conventions by which a computer system communicates with the person operating it.

value: An item of information that can be stored in a variable, such as a number or a string.

variable: (1) A location in the computer's memory where a value can be stored. (2) The symbol used in a program to represent such a location.

wraparound: The automatic continuation of text from the end of one line to the beginning of the next, as on the display screen or a printer.

write: To transfer information from the computer to a destination external to the computer (such as a disk drive, printer, or modem) or from the computer's processor to a destination external to the processor (such as main memory).

Index

Cast of Characters

A

B

C

D

E

Tell Apple About .... *A Touch of AppleSoft BASIC*

Please contact your authorized Apple dealer when you have questions about your Apple products. Dealers are trained by Apple Computer and are given the resources to handle service and support for all Apple products. If you need the name of an authorized Apple dealer in your area, call toll free: 800-538-9696.

Would you like to tell Apple what you think about this product? After you have had an opportunity to use this product we would like to hear from you. You can help us to improve our products by responding to the questionnaire below and marking the appropriate boxes on the card at the right with a #2 lead pencil. If you have more than one response to a question, mark all the boxes that apply. Please detach the card and mail it to Apple. Include additional pages of comments if you wish.

1. What is your overall satisfaction with *A Touch of AppleSoft BASIC*? (**1**=poor . . . **6**=excellent)
2. Did you expect programming material with your computer? (**1**=no, **2**=yes)
3. Would you have tried programming if *A Touch of AppleSoft BASIC* had not been included? (**1**=no, **2**=yes)
4. If you program, why do you program? (**1**=hobby, **2**=for sale, **3**=employer, **4**=education)
5. Which Apple computer are you using? (**1**=Apple IIGS, **2**=Apple IIe)
6. How much prior experience have you had using computers? (**1**=none . . . **6**=extensive)
7. Who in your household used *A Touch of AppleSoft BASIC*? (**1**=male adult, **2**=female adult, **3**=male child, **4**=female child)
8. How much of *A Touch of AppleSoft BASIC* have you read? (**1**=entire manual, **2**=specific sessions)
9. How easy was *A Touch of AppleSoft BASIC* to read and understand? (**1**=difficult . . . **6**=very easy)
10. How would you rate the organization of *A Touch of AppleSoft BASIC*? (**1**=poor . . . **6**=excellent)
11. How helpful were the example programs? (**1**=not helpful . . . **6**=very helpful)
12. Did you work with the book at your computer, doing each example as it was presented? (**1**=none of the time, **2**=some of the time, **3**=most of the time, **4**=all of the time)
13. To what degree did the technical material in *A Touch of AppleSoft BASIC* meet your expectations? (**1**=low . . . **6**=high)
14. What did you like best about *A Touch of AppleSoft BASIC*?
15. What did you like least about *A Touch of AppleSoft BASIC*?
16. Please describe any errors or inconsistencies you may have encountered with the materials. (Page numbers are helpful.)
17. What suggestions do you have for improving *A Touch of AppleSoft BASIC*?

A Touch of AppleSoft BASIC

Please Do Not Mark Above This Line

1. ☐ ☐ ☐ ☐ ☐ ☐
2. ☐ ☐
3. ☐ ☐
4. ☐ ☐ ☐ ☐
5. ☐ ☐
6. ☐ ☐ ☐ ☐ ☐ ☐
7. ☐ ☐ ☐ ☐
8. ☐ ☐
9. ☐ ☐ ☐ ☐ ☐ ☐
10. ☐ ☐ ☐ ☐ ☐ ☐
11. ☐ ☐ ☐ ☐ ☐ ☐
12. ☐ ☐ ☐ ☐
13. ☐ ☐ ☐ ☐ ☐ ☐
14. ______
15. ______
16. ______
17. ______