Logo for Kids: An Introduction

by Bob DuCharme

Table of Contents

To the Adults (and Logophiles)

I began writing a an introduction to programming aimed at middle-school aged kids and decided not to make a complete book out of it, so I'm just putting it on the web for anyone who wants it. It's about 90 pages, introducing kids to the basic concepts of programming using UCB Logo. I decided to use UCB Logo because it runs on PCs and the Mac, and when I had ideas for turning this into a complete book and publishing it, I thought it would have a better chance in the educational market if it used a Logo distribution that worked on both platforms. The fact that UCB Logo is free, and runs under Linux, are also nice bonuses.

The book assumes that UCB Logo is already installed on the computer that its reader will be using and that the reader knows where to find the icon used to start it up. For information on installing it, see http://www.snee.com/logo.

The "Before You Get Started" section following this one is aimed at the book's audience, but you should read it as well to help your young friend decide whether he or she is really ready for the book. Kids who've had any kind of computer class at school and are reading chapter books on their own should be fine.

To answer a few questions some of you might have...

- Why doesn't the book cover the `edit` command, workspaces, and other features that many people feel are integral to Logo? Because they're too Logo-specific. In deciding which parts of Logo to teach, I chose aspects of the language with equivalents in other programming languages so that the reader could more easily move on to other languages. My main goal in writing this book is to get kids interested in programming and ready to move onto other languages, not to convince them of the beauty of Logo—they won't be in any position to judge Logo as a language until they've learned a few other languages, anyway.

- Why doesn't the book have the reader use Jove, the text editor that comes with UCB Logo? Because after using it under Windows 98 and Windows 2000, I decided that the relationship between UCB Logo and Jove was too flaky to use under Windows, even when following the installation's suggestion to create a `logo.bat` file that sets the EDITOR environment variable to point to Jove and so forth. Kids shouldn't have to deal with that flakiness. So, my installation notes show how to point Windows UCB Logo to Notepad instead.

  Much as I love Emacs, if you know a young reader who will be using Logo under Linux, prepare to do a bit more tech support.

- Why do the filenames created as part of the exercises have an extension of `logo` after the filename's period, when other Logo program files that come with UCB Logo don't use extensions? For three reasons: first, because when you tell the Windows Notepad editor to edit a file and only give a filename and not an extension (for example, `flower` instead of `flower.logo`.) it automatically assumes an extension of `txt` if you don't add a period after the filename. When you finish editing the file and Berkeley Logo automatically tries to load it, it won't find it, because it doesn't know about the `txt` extension that got added on. Being explicit with the extension makes this easier.

  Secondly, when the book's readers move on to other programming languages, they'll learn that when a filename has a specific extension to indicate that the file holds source code in a particular language, it makes it easier to know exactly what they can do with do with that file. Simple C, C++, Java, and Perl programs can look quite similar, but if a given file has an extension of `c`, `cpp`, `java`, or `pl`, its purpose is much clearer.

Lastly, I don't want the reader to think that there is some magic relationship between a filename and a procedure name. A `flower` procedure doesn't have to be defined in a file called `flower`; it can be defined in a file called `tree`, `temp.tst`, `x2934732`, or `flower.logo`. I think this last spelling makes the most sense.

Before You Get Started...

- **Do you have UCB Logo already installed?** If not, have someone help you get it by using a web browser to go to http://www.snee.com/logo. This web page explains how you can get UCB Logo off the Internet for your computer.

- **Have you ever done any typing or keyboarding on a computer?** When this book tells you to type a few words on the computer keyboard, it could be very frustrating if it takes a while to find each key. You should already know how to type lower-case and upper-case letters, how to use the spacebar and cursor keys, and where the **Enter** and **Backspace** keys are (or, if you're using a Macintosh, where the **return** and **delete** keys are, because they do the same things). If you don't know, ask your teacher or librarian about programs that can teach you to use the computer keyboard a little better before you learn programming. It's kind of like using a musical instrument—you can't make much music until you know where the notes are!

- **Have you ever used a program where you create something, save it in a file with a name you make up yourself, and then open the file again later?** (You should already know what a file is, too.) For example, perhaps you've drawn a picture on the computer screen or you wrote a story, then saved it, and opened it up later. If not, you may need some help when you save the Logo programs that you create and then open them up to use again later.

- **Remember, there's a glossary in the back of this book.** A glossary is like a dictionary, but instead of defining thousands of words, it just defines the new words in one book. If you see a word in this book and you aren't sure what it means, check the glossary.

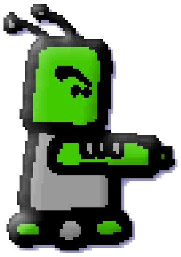
This book is kind of like one of those Lego, Erector Set or K'nex kits. The book gives you pieces of the Logo programming language, just like those kits give you blocks or other building pieces, and it shows you how to put them together into several different projects. After you've built those projects, you'll have a better idea of what the different pieces can do. Then the real fun starts: it's time to make up your own creations!

Robots, Commands, and Turtles

What is Programming, and Why Is It Fun?

Logo is a computer programming language designed to help kids and adults learn programming quickly and easily. But what is a programming language? Why would it be fun to learn one?

Imagine that you have a robot named Ringo. You only know of seven words that Ringo understands: GO, FORWARD, BACKWARD, TURN, LEFT, RIGHT, and FEET. Your robot also understands numbers. When you say "GO FORWARD 3 FEET" Ringo moves forward three feet. When you say "TURN LEFT," Ringo turns to his left. If you say "MAKE ME A SANDWICH," Ringo doesn't do anything. Why? Because none of the words in the instruction "make me a sandwich" are part of Ringo's language—he only understands numbers and the words GO, FORWARD, BACKWARD, TURN, LEFT, RIGHT, and FEET.

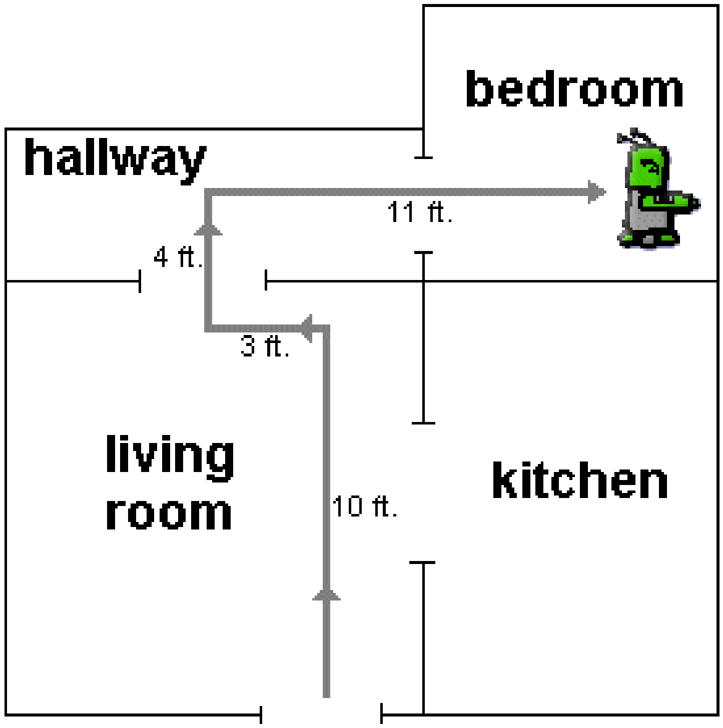


Now, let's say that you like to leave Ringo in the living room by the front door so that you can show him to your friends when they come over. One day, your mom tells you "This room is a mess! Put away your stuff! Ringo belongs in your bedroom!" You don't want to pick up Ringo and carry him, because he's too heavy. You can't say "RINGO! GO TO MY BEDROOM!" because he might understand the word "GO," but he doesn't understand any of the other words.

If you can give Ringo the right directions to your bedroom, though, you can get him to go there himself. So you tell him how to get there with these directions:

- GO FORWARD 10 FEET
- GO LEFT
- GO FORWARD 3 FEET
- GO RIGHT
- GO FORWARD 4 FEET
- GO RIGHT
- GO FORWARD 11 FEET

And he goes!



Ringo goes from the living room by the front door to your bedroom

You know that you must be careful—if you said "GO FORWARD 12 FEET" when you should have said "GO FORWARD 10 FEET," he's going to bump into a wall, and that's not good for Ringo or the wall.

The next day, you play with Ringo, and have him chase the cat around the living room. When you're done, you leave him by the front door because one of your friends from school is coming over, and you want to show Ringo to her. But again, your Mom says: "This room is a mess! Put away your stuff! Ringo belongs in your bedroom!" So again you tell Ringo:

- GO FORWARD 10 FEET
- GO LEFT

- GO FORWARD 3 FEET
- GO RIGHT
- GO FORWARD 4 FEET
- GO RIGHT
- GO FORWARD 11 FEET

And Ringo goes to your room.

The next day, after you play with Ringo, your Mom comes in the living room and says "I'm really getting tired of finding Ringo in here! Put that robot away right now!" You say to yourself, "I'm really getting tired of repeating that long list of directions. Sometimes I get the numbers mixed up, and Ringo bangs into a wall, and Mom gets even madder. I wish I could just say 'Ringo! Go to my room!"

So you look in the little book that came with Ringo, and find out about four more words that he knows: NEW, WORD, START, and END. Using these words, you can teach Ringo new words. When you tell Ringo "NEW WORD," the next word you say after that is the new word you're teaching him. Then, you say "BEGIN" and explain what the new word means. (Remember, when you explain anything to Ringo, you have to stick to words that he already understands.) When you're done explaining, you say "END" and Ringo remembers that the explanation in between when you said "BEGIN" and when you said "END" goes with the new word. For example, imagine if you said this to Ringo to teach him the word "HOME":

- NEW WORD HOME
- BEGIN
- GO FORWARD 10 FEET
- GO LEFT
- GO FORWARD 3 FEET
- GO RIGHT
- GO FORWARD 4 FEET
- GO RIGHT
- GO FORWARD 11 FEET
- END

Now, when your mother says "Get that robot out of the living room!" you just say "HOME!" Ringo knows what "home" means: GO FORWARD 10 FEET, GO LEFT, GO FORWARD 3 FEET, GO RIGHT, GO FORWARD 4 FEET, GO RIGHT, GO FORWARD 11 FEET. And he goes back to your room!

If you did this, you would be programming Ringo. The series of steps are the program, and the "HOME" part is the name of the program you wrote for Ringo. The ***programming language*** is the words that Ringo understands: the eleven words GO, FORWARD, BACKWARD, TURN, LEFT, RIGHT, FEET, NEW WORD, START, and END.

Actually, a language is more than just the vocabulary, or list of words that you can use; it's also the rules for putting those words together. If you told Ringo "FEET GO END 5," he would recognize all the words, but he wouldn't know what to do, because you put those words together in a sentence that he doesn't understand. When you learn any new language, whether it's a computer language such as Logo or a spoken language such as Spanish or Chinese, you learn the words you can use and how to put them together. We call this collection of rules a ***grammar***.

It would be a lot of fun to have a robot obey your instructions and learn new words from you. A computer can't walk from one room to another like Ringo, but it can do many things that a robot can't do. It can make pictures, make music, send messages back and forth to other computers connected to it, and many other things. You may not have a robot, but in your home or school or library there's probably a computer that you can use. Once you learn some instructions that you can give to a computer and learn how to put some of those instructions into a list and then give that list a name, you can teach your computer to do new things. Making your computer do these new things is as easy as sending Ringo to the bedroom by telling him one word. This is what writing and running programs is all about. Every time you play a game on a computer or a Nintendo or Sega machine or look at web pages on the Internet or draw pictures on a computer screen, you're running a computer program that someone wrote. After you read this book, you'll be able to write your own programs!

In our story about Ringo, we had a special little made-up language that we used to give him instructions. What is the language you use to program your computer? What are the words that your computer understands, and how do you put those words together? There are actually many languages to choose from. Some are better for inventing new computer games, some are better for controlling robots, some are better for describing music, and some are better for inventing new computer languages. These computer programming languages often have strange or funny names like Java, C, C++ (pronounced "see plus plus"), C# ("see sharp," like the musical note), Python, Ruby, shell script, assembly language, or Logo. (When the people who make Lego building blocks invented a programming language to control their Mindstorms robots, they called it "RCX Code," and they based it on Logo. So if you learn Logo, you'll have a good head start at learning the language that controls Mindstorms robots as your next computer language.)

Try This!

Do you know any programmers? Maybe your Mom, or Dad, one of their friends, or someone who works with them? Ask them what programming language they like to use and why.

Logo and Berkeley Logo

The ***Logo*** programming language was invented over thirty years ago to make it easier for kids to learn how to program computers. Logo could control a robot that was similar to Ringo the robot, but Logo controlled a robot turtle.

(It was also different from Ringo because someone really made it—Ringo is just a story that I made up for this book.) Instead of walking around someone's home, the robot turtle walked around on a piece of paper. It held a pen on the paper so that it drew lines as it walked around. Instead of writing a program to send the turtle from one room to another, you could write programs to draw pictures. The very first Logo robot turtle was named "Irving."

It was expensive to make robot turtles and hook them up to computers. A new version of Logo was invented where the programs that people wrote with Logo controlled a pretend turtle robot on the computer screen instead of a real robot turtle on a piece of paper. You could still write the same programs to draw pictures, but the picture was on the screen. If you really wanted your picture on a piece of paper, you could send the picture to your computer's printer, just like you do after you draw a picture on a computer screen today.

A ***Logo interpreter*** is a program that lets you give instructions to the pretend turtle robot and then shows you the results of your instructions. You can give it one instruction and see what happens (just like you could tell Ringo the robot "GO FORWARD 3 FEET"). You can also put a bunch of instructions together, save it as a program, and then run the program. A Logo interpreter isn't just about turtles, though; we'll see how you can write programs that make up stories or create computer games with no help from any turtles. People usually start Logo programming by making pictures with the turtle because it's fun and easy, and that's what we'll do.

There are different Logo interpreters that you can choose from. We're going to use one called Berkeley Logo for two reasons: first, it's free. Second, it runs on computers that run the Microsoft Windows operating system, Macintosh computers, and computers that run the Linux operating system, so everyone can use Berkeley Logo.

What's an Operating System?

An ***operating system*** is a special program that starts up as soon as you turn on a computer. You can't do anything else with the computer until the operating system is up and running. In fact, when you turn on a computer and have to wait before you start up your favorite game or other program, that's what you're waiting for. When you click an icon on your screen or pick something on a menu to start up a program, you're actually telling the operating program to start that program up. When you send a picture or a story to the printer, you're really telling the operating system to send it to the printer.

A program that works on a computer with one operating system won't always work on a computer that uses a different operating system. Luckily, Berkeley Logo runs on several different operating systems.

Macintosh computers, which only run the Macintosh operating system, are popular in schools. Programmers often like the ***Linux*** operating system (pronounced "linnix"), and Microsoft Windows is popular in offices and on home computers.

> Knowing how to use different operating systems is like knowing how to play different musical instruments or how to play different sports: it means that you can do more, and you can have more fun!

The "Berkeley" in "Berkeley Logo" is the University of California at Berkeley, the school where Berkeley Logo was invented. (Sometimes the program is called "UCB Logo.") The study of programming languages, operating systems, and other parts of building and using computer programs is called ***computer science***. A lot of the most important work in the history of computer science took place at the University of California at Berkeley.

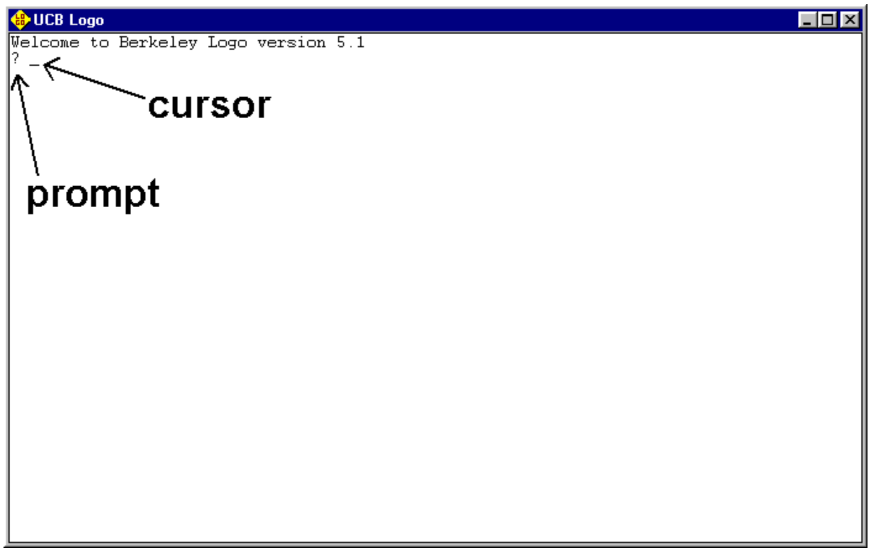
Let's start up Berkeley Logo.

* **Whenever you see words that look like this in this book, that means that they describe something for you to do. Follow the instructions, and you'll make Berkeley Logo do the things that you see it doing in the pictures in the book.**

Double-click the Berkeley Logo icon to start up this Logo interpreter program. (If you are using the Windows operating system, you may need to click a different icon called "logo.bat" to start up Berkeley Logo. Ask whoever helped to set up Berkeley Logo on your computer.)

On a Macintosh, you'll see two windows open up. One is where where your turtle will draw pictures, and the other window, which says "Welcome to Berkeley Logo" in it, is where you type instructions to Logo. If you can't see both Logo windows on your Macintosh, one might be in front of the other; try moving the one you do see to the side. On a Windows or Linux computer, you'll just see the "Welcome to Berkeley Logo" window when you start up Logo.

The little question mark in the white window is called a ***prompt*** or ***command prompt***. This is how most computers show you that they're ready for you to enter an instruction. The underscore line (_) next to the bottom of the question mark is called the ***cursor***. The cursor shows where your letters, numbers, and spaces will appear on the screen when you type them on the computer keyboard. In our story about Ringo the Robot, we used our voice to give Ringo instructions, but we'll give instructions to our Logo turtle by typing them on the keyboard.



```
Welcome to Berkeley Logo version 5.1
? _
```

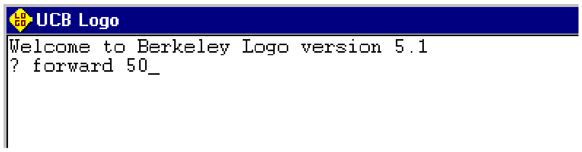
Opening UCB Logo screen

Giving Instructions to the Turtle

Let's enter an instruction:

* **Tell the turtle to take 50 steps forward by typing the instruction below. (Whenever you see letters `that look like this`, like `forward 50` below, it's something for you to type on your computer's keyboard.)**

```
forward 50
```



```
Welcome to Berkeley Logo version 5.1
? forward 50_
```

Entering your first instruction

Before you press the **Enter** key, here are a few things to remember about typing in an instruction that tells Logo to do something:

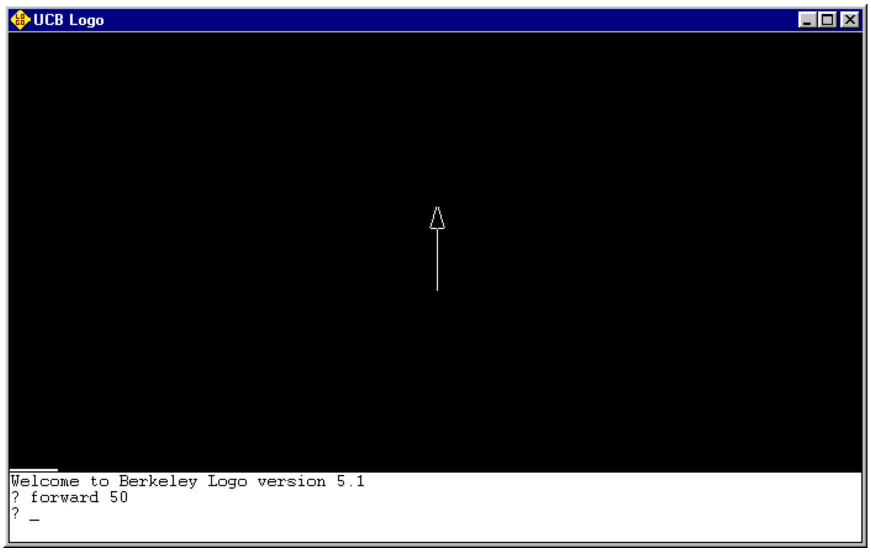
- Type any spaces just as you see them in the book. For the instruction above, that means you have to type a space after the "d" in "forward." If you forget this space, you'll type "forward50," which is not something that Logo understands.

- If you make any typing mistakes, you can press the **Backspace** key (or, on a Macintosh, the **delete** key) to move your cursor back to the mistake and fix it. On PCs, the **Backspace** key might have the word "Backspace," or it might have an arrow pointing to the left. The cursor key that moves your cursor to the left will also have an arrow pointing to the left, but if there's one on your **Backspace** key it will be bigger than the one on the cursor key.

- Logo is not ***case-sensitive***. That means it doesn't care whether you type instructions using upper-case letters, lower-case letters, or a combination of upper- and lower-case. You could type the instruction above as "FORWARD 50" or "FoRwArD 50" or "forWARD 50" and it would work just as well.

- The word ***instruction*** has a special meaning in Logo. The Logo programming language has various special words, like the word `forward` and others that we'll learn soon. An instruction is a combination of one or more special words and the information that they need to do their job like the number "50" above. Usually, a complete line that you type in to Logo (like `forward 50`) is an instruction, although an instruction can be spread out over multiple lines, and you can also put multiple instructions on one line.

* **Press the Enter key. This tells UCB Logo to *execute*, or run, the typed instruction. (Macintoshes call this key the return key, so if you're using a Mac, then whenever this book tells you to press the Enter key, press your return key.)**



```
Welcome to Berkeley Logo version 5.1
? forward 50
? _
```

The turtle responds to your instruction

Your turtle shows up as a little triangle pointing up. It moves forward 50 steps, drawing a white line on the black background as it goes. It probably appeared and moved forward so quickly that you didn't even see it move. But where did it first appear?

- **On a Windows computer, you only have one Logo window, so part of that window goes black and the drawing happens there.**
- **On a Macintosh, when you first started up UCB Logo, two windows appeared: the one where you type your instructions and the one where the turtle draws. After you type your instruction to the turtle in the instructions window you'll see the result in the other window.**
- **On a computer running Linux, there is a separate window for turtle drawing, as with the Macintosh, but this window doesn't appear until Logo needs it. So, when you execute the command above, the second window suddenly appears and the line gets drawn there.**

The `forward 50` instruction that you typed also ***scrolled*** up a line. That means that it moved up on the screen just as if it were on a roll of paper that unrolled a little so that the next instruction could be typed on a blank line under it. As a matter of fact, thirty years ago, this is how most computers really worked—instead of a ***monitor*** (that's the part of your computer that looks like a TV set) they had a printer that kept unrolling paper to show the instructions that you typed at the keyboard and the computer's responses to those instructions.

The line that the turtle drew on your screen wasn't very long. If the turtle took fifty steps, they must have been tiny steps. If you want to draw longer lines, you can use much bigger numbers in your instructions. For now, don't enter any instructions unless this book tells you to. You don't want to accidentally send your turtle somewhere where you don't know how to get it back! There will be a chance to play after you learn a few more instructions.

Commands and Instructions

The `forward` part of what you typed is called a ***command***. As we'll see later, certain commands, all by themselves, don't make much sense to Logo—for example, whenever you tell the turtle to move forward, you have to tell it how far to move forward. This extra information that some commands need are called ***parameters*** (pronounced "puh-ram-ih-terz.") When you put together a command with any parameters that it needs, you have a complete instruction for Logo to follow.

Just like Ringo the Robot, the turtle understands instructions that tell it to turn. Unlike Ringo, you don't have to use the word "turn" with the turtle—just say "left" or "right" and it knows what you mean. Let's try it.

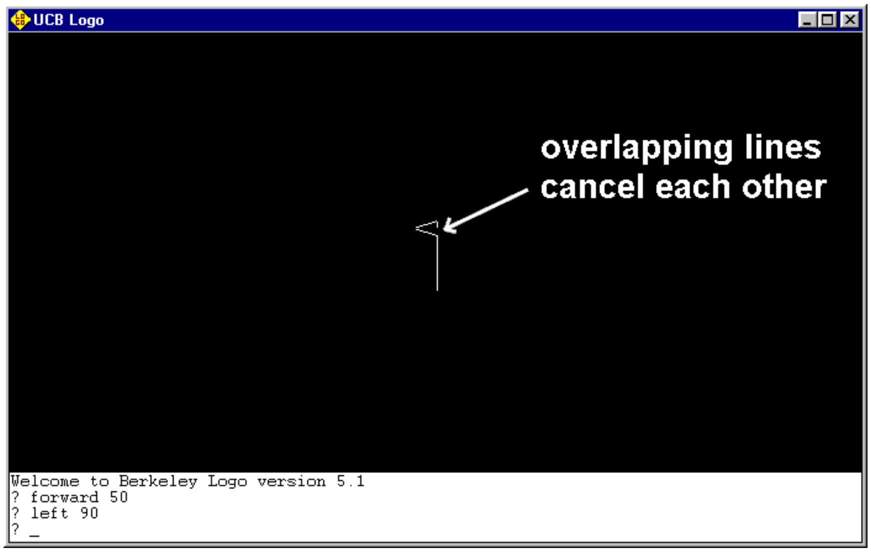
* **Enter this instruction:**

```
left 90
```

90 what? Not 90 steps. The distance your turtle turns is measured in something called "degrees," just like a thermometer measures the temperature in degrees. (It's the same word we use to measure temperature with a thermometer, but it means something different here.) The `left 90` instruction tells the turtle to turn left 90 degrees.

* **Execute the instruction. (That is, press Enter).**

The triangle turns, and then part of the triangle overlaps with the line that the turtle drew with your first instruction, so the two white lines cancel each other out and show up as a blacked-out part of the two lines. The two instructions that you've typed both scroll up in their window by one line.



```
Welcome to Berkeley Logo version 5.1
? forward 50
? left 90
? _
```

The turtle turns left

Try This!

Degrees measure turning as part of a circle. If you stand up and face your computer and then turn all the way around so that you're facing your computer again, you turned a whole circle. You turned 360 degrees. If you turn again, but this time only turn halfway around that circle so that you're facing away from your computer, you turned around for 180 degrees, or half of 360. (If you ever hear of a car doing a 180 or doing a 360, this is what they're talking about: it spun halfway around or all the way around. This is not a safe thing for a car to do!)

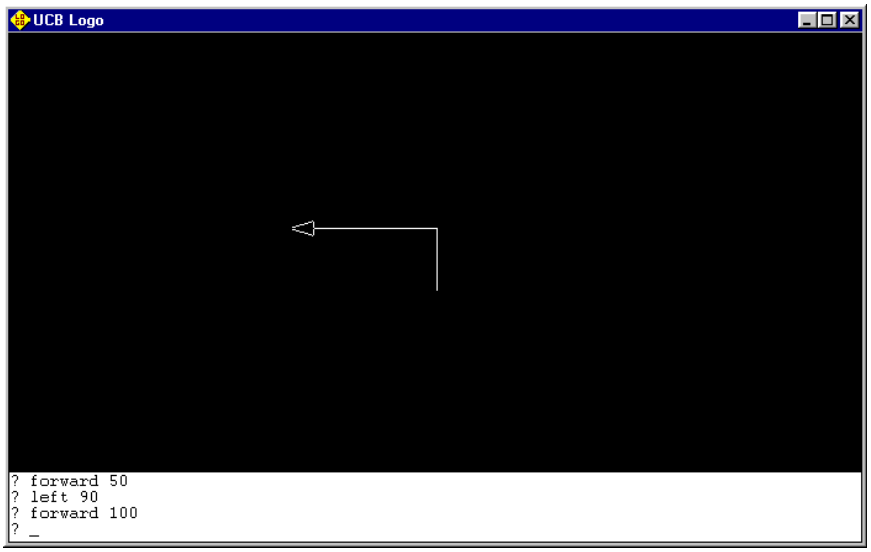
Half of 180 is 90, so a simple left turn or right turn to go around a corner is a 90 degree turn, and that's the kind of turn that you just told your turtle to do.

Let's tell the turtle to take some more steps now that it's facing in a new direction.

* **Enter this instruction and execute it:**

```
forward 100
```

The turtle moves twice as far as it did before. This time, it moves to the left, because that's where it was pointing when you told it to move forward.



```
? forward 50
? left 90
? forward 100
? _
```

The turtle moves to the left

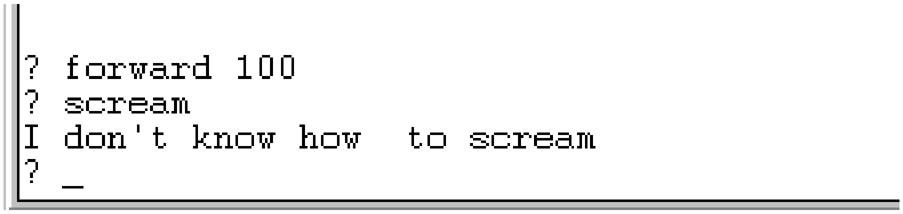
Error Messages

So far you've given your turtle two instructions, and it understood both: "forward" and "left." What happens if you give it an instruction that it doesn't understand? For example, it doesn't know how to scream. Let's try telling it to.

* **Enter the following instruction and then execute it:**

```
scream
```

Logo puts a message on the screen to answer you:



```
? forward 100
? scream
I don't know how  to scream
? _
```

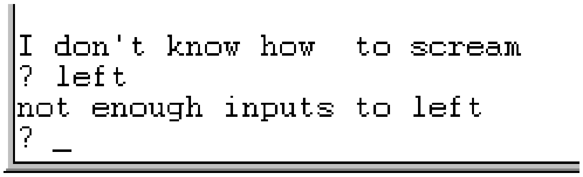
Logo doesn't know how to scream!

This is how Logo tells you that it doesn't understand something. If you had made a mistake when you typed "forward" in your first instruction and typed "forwarg" instead, Logo would have told you "I don't know how to forwarg." This is called an ***error message***. If you ever see one of these error messages when you didn't expect it, double-check what you typed to make sure that you typed it perfectly.

What if you told the turtle to turn left, but you didn't tell it how far to turn? Let's see what happens.

* **Enter this instruction with no number and execute it to see what happens:**

```
left
```



```
I don't know how  to scream
? left
not enough inputs to left
? _
```

Left? How far left?

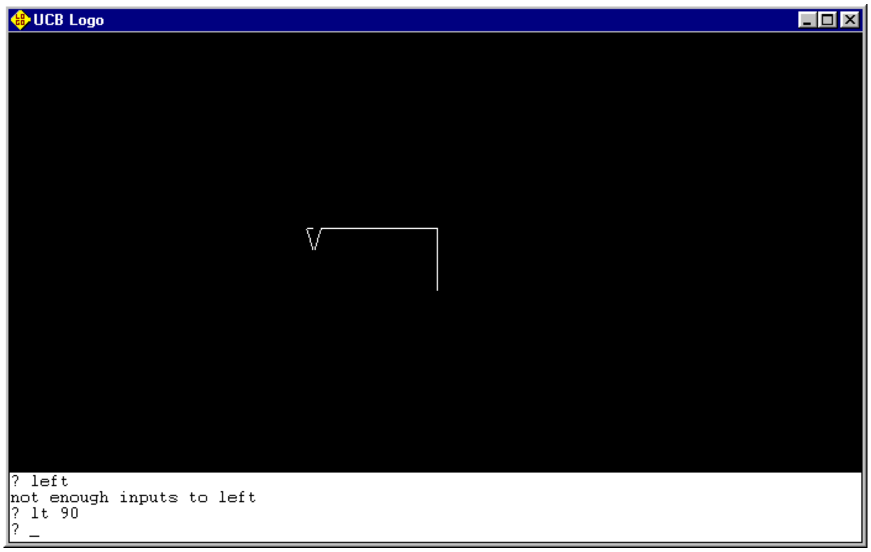
It tells you "not enough inputs to left." (The word ***input*** refers to information coming into a computer. Input might come from a file on a disk, or from another computer hooked up to yours by a phone line, or from a microphone that you're singing into, or from a keyboard that you're typing on.) "Not enough inputs" means that you didn't type enough on your keyboard for Logo. There were "not enough inputs" to the `left` command; as we saw before, this command needs a number after it to tell the turtle how far to turn left. (Another important computer word is ***output***, which describes information coming out of a computer. It might be coming out onto your monitor screen, or onto your printer, or out of a speaker attached to your computer, or out over a phone line hooked up to another computer.) Other computer languages usually call the extra pieces of information that certain commands need to do their job "parameters," a term we learned when we entered our very first instruction to Logo earlier in this chapter.

More Turtle Drawing

Let's try the `left` command again. This time, we'll use the special short way to type it that leaves out the second and third letters: `lt`.

***   Enter the following in the input box and execute it:**

```
lt 90
```



```
? left
not enough inputs to left
? lt 90
? _
```

Turning left with the abbreviation for "left"

(Most Logo commands have a special shorter version, so you can do a lot with Logo with just a little typing.) Because the cursor was pointing to the left when you executed this instruction, a 90 degree left turn now has it pointing down.

Having the turtle draw white lines is getting boring. Let's tell it to change the color.

* **In the next instruction, the phrase "set pen color" is three words, but the Logo command `setPenColor` is only one word, so the only space you'll type in this instruction is the one between the "r" and the "4." Enter this instruction and execute it:**

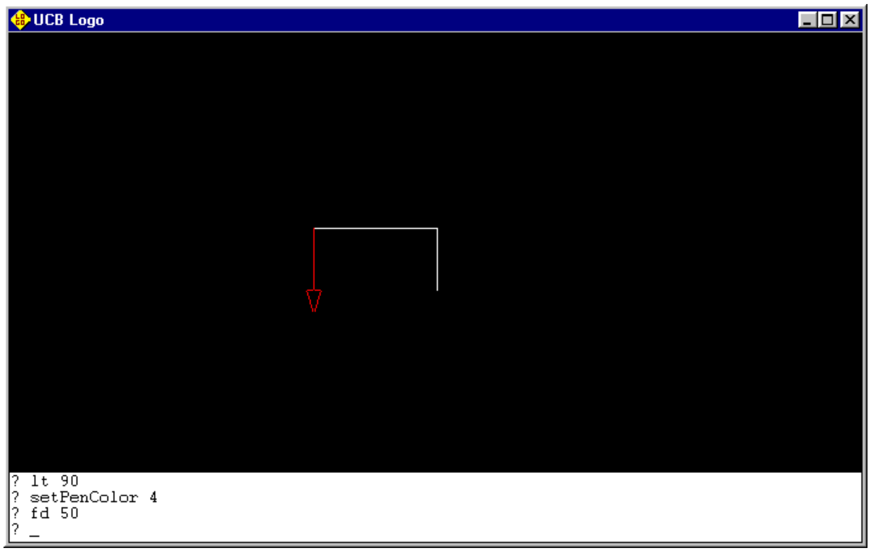
```
setPenColor 4
```

As we saw with your first instruction, the case doesn't matter when you type Logo instructions. You could type "setpencolor" or "SETPENCOLOR" or "SeTpEnCoLoR" and it would still work. With a lot of computer words that are made up by joining words together, people sometimes write the first letter of all the words after the first one in an upper-case letter to make it easier to read. (isthiseasiertoread, orIsThisEasierToRead?) Many computer languages do this all the time, and it's a good habit to get into, so that's why I do it in this book.

* **Draw a line and see how it looks. Like the `left` command, the `forward` command has a short form that can save you some typing: `fd`.**

```
fd 50
```

The line shows up red!



```
? lt 90
? setPenColor 4
? fd 50
? _
```

Drawing a line with the new pen color

You set the pen color with the `setPenColor` command to color number 4, which is the color red. You then entered the `forward` command, but you entered it as `fd` because it means the same thing to Logo, and saves you some typing. When the turtle moved forward, it drew a red line instead of a white one.

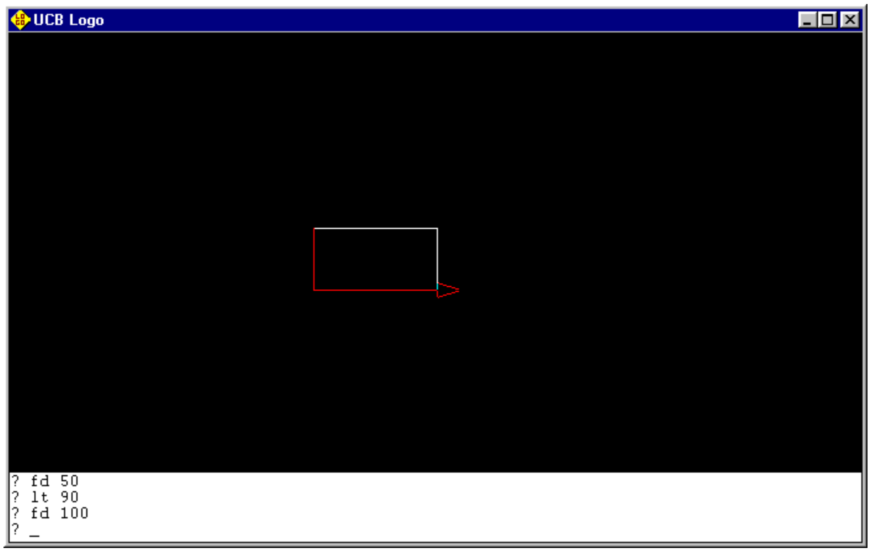
Later in this chapter, you'll see a list of 16 different color numbers and you'll get to play with them to draw whatever you like.

Let's finish the white and red rectangle.

***   Enter and execute the following two instructions:**

```
lt 90
fd 100
```

Your rectangle is complete!



```
? fd 50
? lt 90
? fd 100
? _
```

The finished rectangle

Picking Up and Putting Down the Pen

Now we're going to draw a little square. We don't want to draw it right under the rectangle, but 40 turtle steps away from it.

So far, the turtle has drawn a line every time it's moved. How can we get it to move without drawing a line? By telling it to pick the pen up before moving.

* **Use the `penUp` command to tell the turtle to pick up its pen:**

```
penUp
```

* **If we tell the turtle to turn left again, it will point to the top of the screen. We want it to point down to the bottom, because the new square will be below the rectangle, so we use the `right` command to tell the turtle to turn right 90 degrees:**

```
right 90
```

* **Tell it to take 40 turtle steps in the direction it's facing:**

```
fd 40
```

The pen is up, so the turtle doesn't draw anything, but the triangle moves down.

* **Use the `penDown` command to put the pen back down so that the next moves draw more lines:**

```
penDown
```

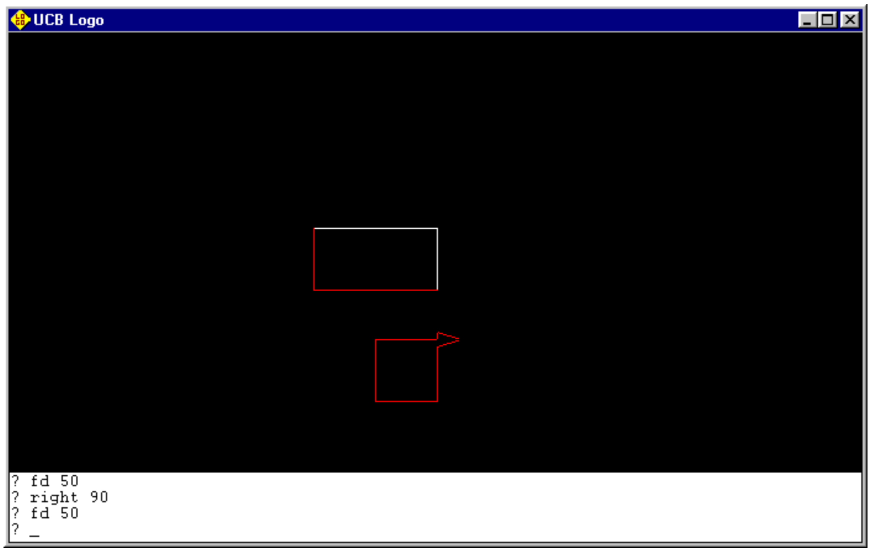
* **Draw a square by repeating the following two instructions four times:**

```
fd 50
right 90
```

Some programming languages don't let you enter multiple instructions on the same line. Logo does, so you don't have to press Enter after each instruction. If you wanted to, instead of entering the two lines above four times, you could have entered the following single line four times:

```
fd 50 right 90
```

Entering the two commands four times drew a red square:



```
? fd 50
? right 90
? fd 50
? _
```

A red square under the rectangle

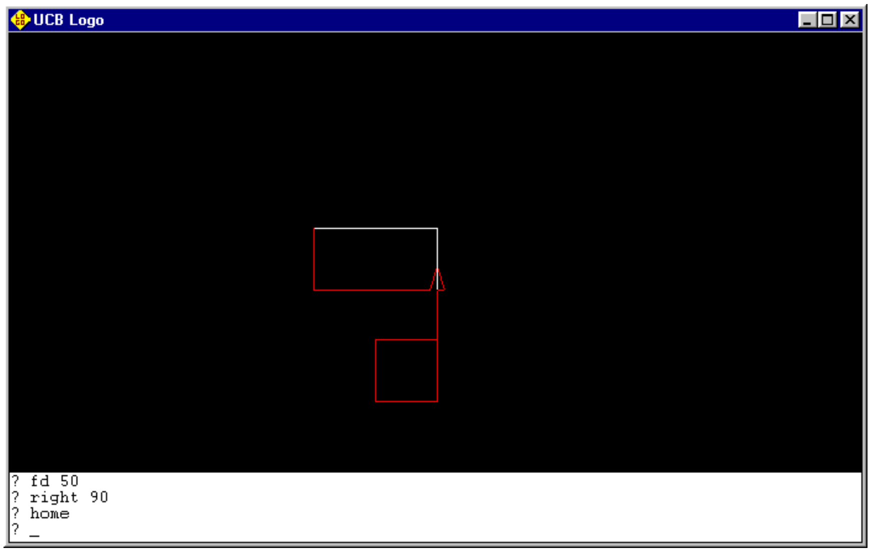
Cleaning Up

The `home` command sends the turtle back to where it started.

*   **Enter the following and press Enter:**

```
home
```

The turtle jumps back to the middle of the screen and points to the top of the screen, where it was when you first started up Logo.



```
? fd 50
? right 90
? home
? _
```

The turtle returns home

Because the turtle's pen was down when it jumped home, it drew a line as it went.

The `clearScreen` command clears away everything on the screen and sends the turtle home.

*   **Move the turtle away from the center of the screen:**

```
fd 50
```

*   **Clear the screen and send the turtle home to the center with one command.**

```
clearScreen
```

(For even less typing, you could type the short form of `clearScreen`: `cs`.)

All the drawing on the screen is cleared off.

Try This!

Now draw your own picture! Use any of the commands you've learned so far. Here is a list, along with the short forms you can use if you want to save some typing:

Command	Abbreviation
forward	fd
left	lt
right	rt
setPenColor	setpc
penUp	pu
penDown	pd
home	(no abbreviation)
clearScreen	cs
bye	(no abbreviation)

The end of the list has one new command that you haven't seen before: `bye`. It's like saying "good-bye" to Logo. Enter this instruction when you're all done with Logo, just like you might click on "Exit" or "Quit" in another program.

When you want to set the color of your pen, use this chart to figure out what number to put after the `setPenColor` (or `setpc`) command:

0	black
1	blue
2	green

3	aqua
4	red
5	magenta
6	yellow
7	white
8	brown
9	light brown
10	pea green
11	grayish blue
12	salmon
13	light purple
14	orange
15	gray

Try sending your turtle so far forward that he goes off the screen. What happens?

Running Someone Else's Program

In the story about Ringo the Robot you programmed Ringo so that he would go to your room whenever you told him the word "home." We've seen that "home" is already a command in Logo, sending your turtle to the center of the screen. You can still make up a new command for your turtle by saving a list of instructions using commands that you know, giving the list a name, and then telling the turtle to execute the whole list of instructions by typing in the list's name. The list's name will be a new command for Logo. In fact, all computer commands, like Logo's `forward` and `left` and `penUp` commands, are really running programs. In the next chapter, we'll learn how to make up new commands. Before we do, though, let's learn how to run a program that comes with Logo.

* **You're going to run a Tic-Tac-Toe game program called "ttt". First, try to start it up even though you haven't loaded the program by entering this:**

```
ttt
```

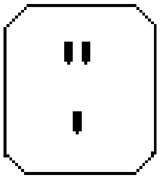
Logo tells you "I don't know how to ttt." Let's tell it to load the `ttt` program so that it knows what the instructions that make up this program are.

* **Enter the following instruction to load the `ttt` program from the `csls` folder (also known as the `csls` subdirectory):**

```
load "csls/ttt
```

If you try to name a file that doesn't exist, Logo will tell you "File system error: I can't open that file," so make sure to spell the folder and command name exactly as they're shown here, with the slash between them. (The slash character is on the same key as the question mark.) If you ever get this error message when trying to open one of your own files, it doesn't always mean that the file you created no longer exists; it probably means that the name you just typed is slightly different from the name that you originally gave to the file, and that no file with the name you just typed exists.

Also, don't forget the quotation mark before the "c." A quotation mark, which programmers also call the "*double quote*," is on the same key as the apostrophe, or "*single quote*":



The key on your keyboard with the quotation mark (or "double quote") and the apostrophe ("single quote")

You'll need to press your Shift key to type a double quote instead of a single quote, just like when you type an upper-case "A" instead of a lower-case "a."

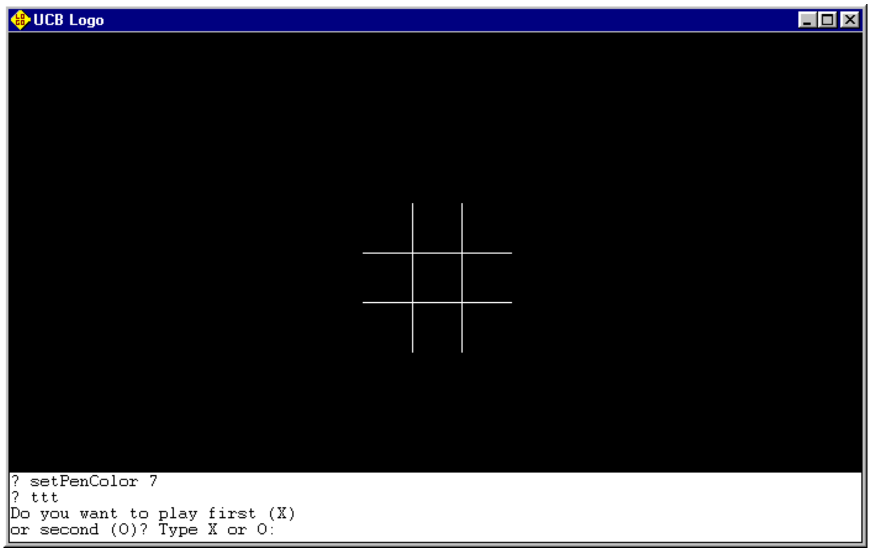
* **If your current drawing color isn't white, and you want the tic-tac-toe game to show up in white on the black background, set the pen color to white before starting up the `ttt` program:**

```
setPenColor 7
```

* **Now try the `ttt` command again.**

```
ttt
```

This time Logo understands the `ttt` command, and it starts up a tic-tac-toe game:



```
? setPenColor 7
? ttt
Do you want to play first (X)
or second (O)? Type X or O:
```

The tic-tac-toe game starts

* **At the bottom of the screen, it tells you to type the letter "X" if you want the first turn in the tic-tac-toe game and "O" (the letter, not the number zero) if you want to go second. Type this (don't press Enter, because the game is only expecting you to type one key):**

```
x
```

* **To tell it where you want to make your first move, enter the number of the square where you want your X by entering a number from 1 to 9.**

1	2	3
4	5	6
7	8	9

The numbers to use when picking your move

* **As soon as you enter a number to show where to put your X, the `ttt` game puts an O in one of the other squares and asks you to make your second move. Continue playing until you or the `ttt` game wins. (It's awfully hard to beat it.)**

When you're done with your first tic-tac-toe game, you have three choices:

- You can enter `ttt` to play another game.
- You can enter "bye" to tell Logo that you're done with it for now. The Logo window will close up.
- You can use the other commands you've learned to keep on drawing. The `ttt` game turned off the triangle that shows where the turtle is, but you can turn it back on with the `showTurtle` command. Even if you don't enter this command, all of the drawing commands you've learned will still work.

In the next chapter, you'll learn how to create and run your own programs just like `ttt`. Yours will start off simpler, but you'll learn how to add more and more to them until you can create your own games!

What We Learned

In this chapter, we learned:

- What programming is.
- What a programming language is.
- What a Logo interpreter is.
- How to start Berkeley Logo.
- How to give instructions to the Logo turtle to make it move around and draw lines.
- How to change the color that the turtle uses to draw.
- What happens when you enter a command that Logo doesn't understand.
- How to tell the turtle to pick up and put down the pen.
- How to clear the screen and send the turtle home to the middle of the screen.
- How to load and run the tic-tac-toe program that comes with Berkeley Logo.

- How to quit out of the Berkeley Logo program.

Don't forget about the glossary in the back of this book. If you see a computer word and you forgot what it means, it's probably explained in the glossary.

New Commands in this Chapter

(See the "Procedures Reference" Appendix in the back of the book for brief descriptions of what these do.)

- `bye`
- `clearScreen`
- `forward`
- `home`
- `left`
- `penUp`
- `right`
- `setPenColor`

More Things to Try

1. List some ways that your Logo turtle is like Ringo the Robot. How is it different?
2. Have your turtle draw a green square that's 100 turtle steps high and 100 turtle steps wide.
3. Do you know how many turtle steps high your screen is? How wide it is? What would be a good way to find out?
4. You've already drawn a rectangle and some squares with the turtle. Can you draw a triangle? How about a star?

Writing and Running Programs

Programmers hate typing. We saw in the last chapter that commands like "forward" and "clearScreen" have abbreviations such as "fd" and "cs" that let you enter those commands with less typing.

Programmers especially hate typing the same thing over and over. Computers are supposed to be good at doing the same thing and over; why should you type "fd 50" and "right 90" four times to make a square? There should be a way to say "do these two instructions four times," and there is: loops.

Loops

You can tell Logo to repeat one or more instructions over and over by using the `repeat` command. After the `repeat` command, enter the number of times to repeat the instructions. Then, between a pair of square brackets, put the list of instructions to repeat. Let's try it.

* **First make sure that your screen is clear and that your turtle is at the center of the screen pointing up. Enter the abbreviation for the `clearScreen` command:**

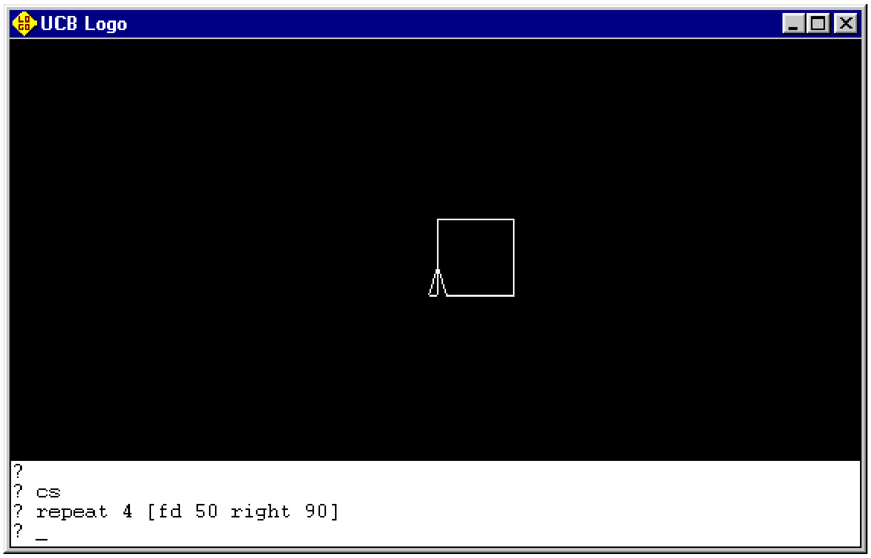
```
cs
```

* **If you can't see the turtle's triangle because the `ttt` program hid it with the `hideTurtle` command, tell Logo to show it with the `showTurtle` command:**

```
showTurtle
```

* **We want to draw a square the same size as the one you drew in the last chapter, but without typing "fd 50" and "right 90" four times. You could do it by typing just one line, but note how this line has two characters that you've never typed before: the square bracket characters (`[` and `]`) are probably on the right side of your computer keyboard near the "P" key. (Programmers often use the word *character* to describe a particular letter, number, or punctuation mark.) Enter the following instruction and execute it:**

```
repeat 4 [fd 50 right 90]
```

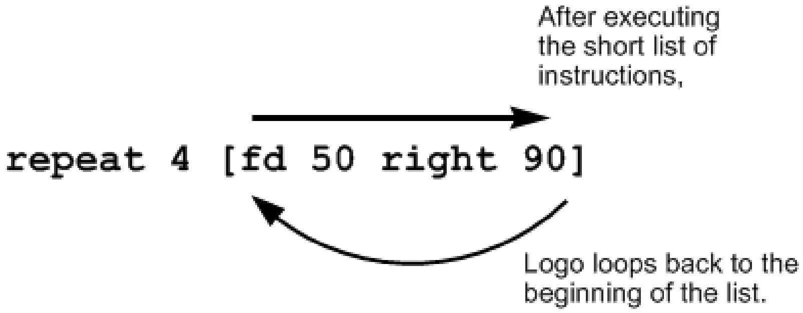


```
?
? cs
? repeat 4 [fd 50 right 90]
? _
```

Drawing a square with a loop

You drew a square by entering one line! ("***Code***" is how programmers refer to the instructions they type.

Programmers use the word ***loop*** to describe a list of instructions that get repeated, because the computer executes the instructions from the beginning of the list to the end and then loops back to the beginning. Instead of square brackets, some programming languages use other ways to show the beginning and end of the list of instructions to repeat. For example, some use curly braces ({ and }) and some use the words "begin" and "end."



```
repeat 4 [fd 50 right 90]
```

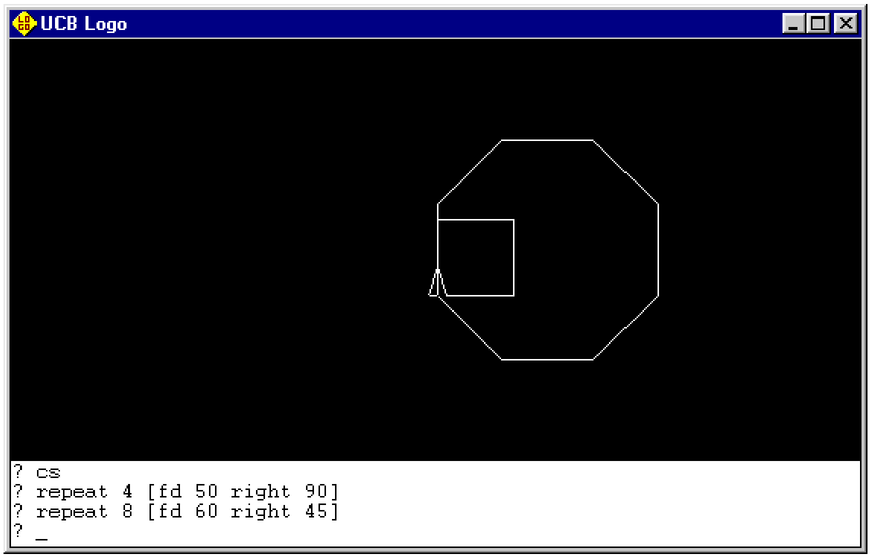
After executing the short list of instructions,
Logo loops back to the beginning of the list.

Repeating a loop's instructions

Let's try another loop. This time we'll draw an octagon, a shape with eight sides.

* **Enter the following and execute it:**

```
repeat 8 [fd 60 right 45]
```



```
? cs
? repeat 4 [fd 50 right 90]
? repeat 8 [fd 60 right 45]
? _
```

Drawing an octagon with a loop

If you live in the United States, can you think of a road sign that has this shape?

Try This!

What other shapes can you draw with a loop? Can you make a star? Try changing all three numbers in the instruction to different numbers and see what the turtle draws. Use the `clearScreen` (`cs`) command if your screen gets messy.

What kind of instructions can you put inside of a loop? Almost any—even another loop! To draw a square, you just used a loop to execute the same two instructions four times. Let's put that whole thing inside of another loop that repeats five times, so that your turtle draws five squares.

* **Clear the screen and send the turtle home:**

```
cs
```

* **Enter and execute the following instruction.**

```
repeat 5 [repeat 4 [fd 30 rt 90] penUp fd 40 penDown]
```

This tells Logo to do the following four steps five times:

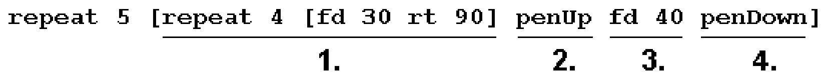
1. **Draw a square. (This step is actually two steps repeated four times: `fd 30` and `rt 90`. But, just like it was one instruction when you executed it before, it's just one step of what's going on five times in the bigger loop.) The square is a little smaller than the last square you told the turtle to draw because we want to fit five of them on the screen.**

2. **Pick the pen up.**

3. **Move forward 40 turtle steps. Because the square is 30 turtle steps high, this moves a little above the top of the square that just got drawn.**

4. **Put the pen back down so that the next move draws a line. Because Logo will loop back to the beginning of the list that gets repeated, that next move will be the beginning of the drawing of the next square.**

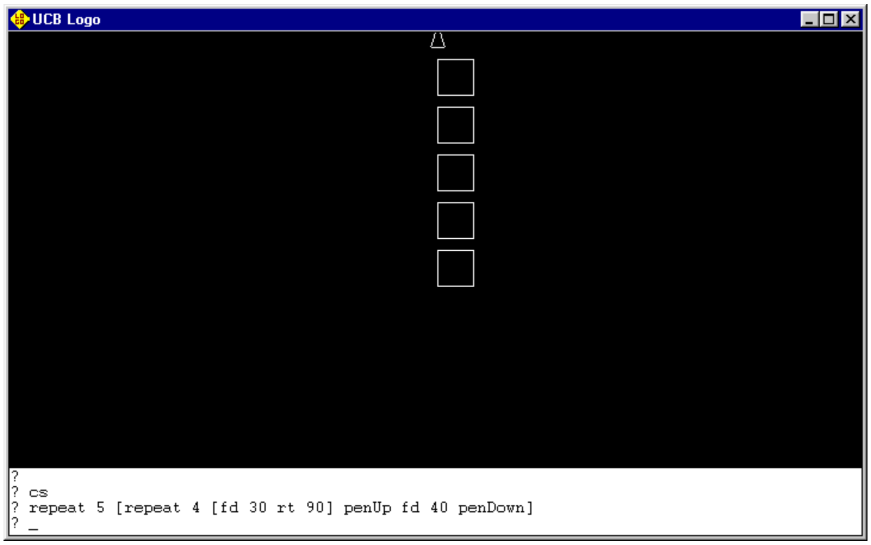
You can tell which instructions get repeated five times because they're the ones inside the pair of square brackets that come right after the "5."



```
repeat 5 [repeat 4 [fd 30 rt 90] penUp fd 40 penDown]
```

The four steps that this instruction repeats five times

Logo draws five squares, with each new one above the previous one. Your turtle moved so high up that it may have gone a little bit off the screen:



```
?
? cs
? repeat 5 [repeat 4 [fd 30 rt 90] penUp fd 40 penDown]
? _
```

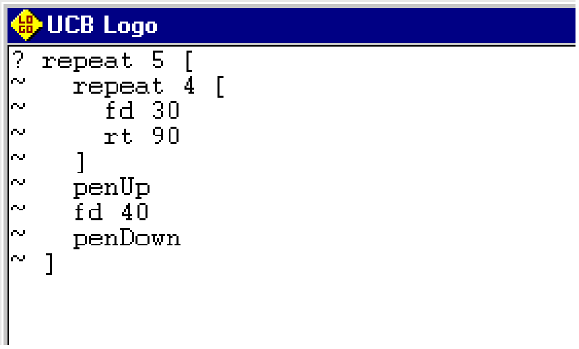
Drawing five squares using a loop inside of a loop

That was a pretty complicated instruction. It was probably hard to read it and understand what it was doing when you first saw it. If we split it up onto many lines, so that each individual step is written on its own line, it's easier to read. This is how programmers usually write complicated instructions like this. They also ***indent***, or add white space before certain instructions to move them over. This way, the lines that go together are grouped together when you look at them (you don't have to type this):

```
repeat 5 [
  repeat 4 [
    fd 30
    rt 90
  ]
  penUp
  fd 40
  penDown
]
```

When the instruction to draw five squares is written this way, the two steps that draw a single square (`fd 30` and `rt 90`) are indented the most, and the four steps that set up the turtle to draw each of the five squares are also indented the same amount.

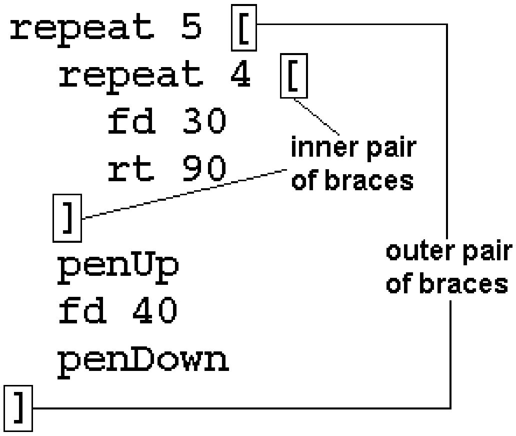
If you did type in the instruction like that, a tilde symbol (which looks like ~ and whose name is pronounced as "till-duh") shows up on each line after the first line because Logo is waiting for you to finish the instruction.



```
? repeat 5 [
~   repeat 4 [
~     fd 30
~     rt 90
~   ]
~   penUp
~   fd 40
~   penDown
~ ]
```

Using multiple lines to enter the instruction

Why does it put the tilde there instead of a question mark? In other words, how does it know that you're not done? Because square braces always work in pairs, so after you enter a left square brace ([) it knows that you're not done until you enter the right square brace (]) that goes with it. Each right square brace goes with the last left square brace that needs a partner. Because of this, Logo knows that the] right after "rt 90" isn't the end of the entire instruction; it's the one that goes with the [from just before the "fd 30" instruction. When that last] gets entered, all the brace pairs that got started have been finished, and Logo executes the instruction.



```
repeat 5 [
  repeat 4 [
    fd 30
    rt 90
  ]
  penUp
  fd 40
  penDown
]
```

Square braces always come in pairs

Try This!

If you haven't already, send the turtle home, clear the screen, and try the new way to enter the instruction that draws five boxes.

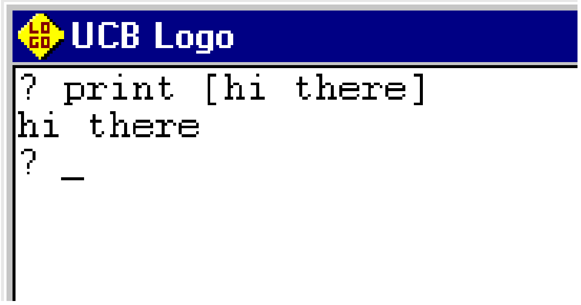
Making Words Appear

When you do fancier things with Logo, it gets to be more and more typing. In our story about Ringo the Robot, what did you do when you got tired of telling Ringo a long list of instructions to send him from the living room to your bedroom? You programmed him. You told him a list of instructions to perform when he heard the special word that you picked to be the name of that list of instructions—in other words, when he heard the name of the program, he executed the program. The name of the program became a new part of Ringo's language.

In the last chapter, when you told Logo to scream, Logo replied that it didn't know how to scream. Now we're going to teach it how to scream. First, we'll learn a new command that makes words and numbers appear in the same scrolling text area where you've seen the instructions and Logo messages appear.

* **Enter and execute this instruction. It uses the same square brackets that you used to show where a list of loop instructions began and ended, because it's a list of two words to put on the screen.**

```
print [hi there]
```



```
? print [hi there]
hi there
? _
```

Telling Logo to display words on the screen

Print? But It's Not Printing!

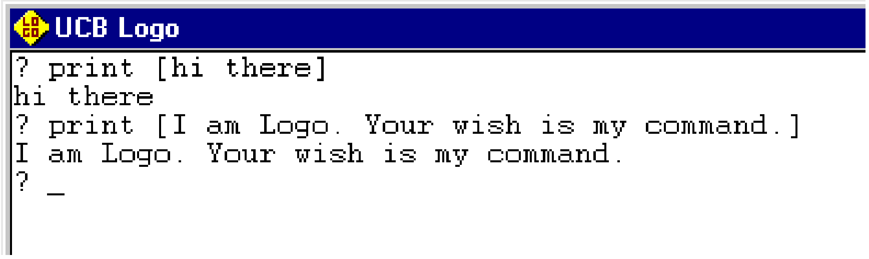
Why is this command called "print" if the words you tell it to print show up on the screen instead of on the printer? Because when Logo was first invented, this command did send the part between the square brackets to a printer.

In the last chapter we learned that a long time ago, computers didn't have monitors (the part of your computer that looks like a TV set). They just had a keyboard and a printer with a roll of paper. When you typed instructions, and when Logo answered your instructions, the printer printed each instruction and response on the roll of paper. Then it unrolled, or "scrolled up," the paper just enough to leave room for the next line that it would print. If you entered `print [hi there]` on a computer with Logo thirty years ago, it would have sent the words "hi there" to the printer.

Because monitors have replaced printers for showing instructions and the result of those instructions, the `print` command sends the words to the computer screen instead. Sometimes we say that it prints the words on the screen, even though it's not really printing with paper and ink.

* **Let's tell Logo to show more words. Enter and execute the following instruction:**

```
print [I am Logo. Your wish is my command.]
```



```
? print [hi there]
hi there
? print [I am Logo. Your wish is my command.]
I am Logo. Your wish is my command.
? _
```

Printing a longer message on the screen

Try This!

Tell Logo to print some more words on the screen. Remember to surround the words with the square brackets.

Your First Program

Let's teach Logo to scream. You'll do this by telling it "Logo, in order to scream, print [Aaaaaaaaaaaeeeeeeeee!]. That's the end of my instructions." This way, you'll be telling Logo how to scream in its own language.

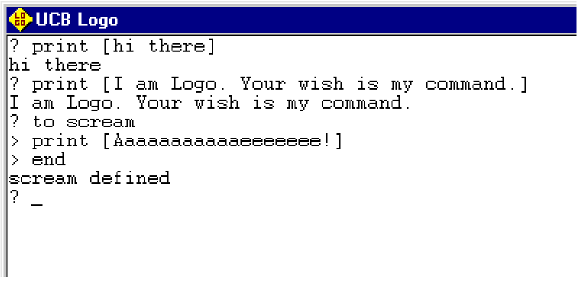
Well, to be honest, not all of that is in Logo's language. We'll leave out the parts that Logo doesn't understand, which means less typing for you to do.

* **Enter the following three lines and press Enter after each one:**

```
to scream
print [Aaaaaaaaaaaeeeeeee!]
end
```

(You can make the scream longer if you want, but make sure that the whole instruction fits on one line.)

Logo knows that the word `to` starts the definition of a new program, and it knows that every line that you enter after that is part of the program until you enter the word `end`. Until you reach the end, Logo displays the prompt as a greater-than sign (>) to show you that it's not waiting for a single instruction that it will execute for you, like it does with the question mark prompt. It's waiting for you to enter `end` to show it that you're done entering your new program. When you reach the `end` line and press **Enter**, Logo should tell you that you've defined a new word in its language: `scream`.



```
? print [hi there]
hi there
? print [I am Logo. Your wish is my command.]
I am Logo. Your wish is my command.
? to scream
> print [Aaaaaaaaaaaeeeeeee!]
> end
scream defined
? _
```

Defining a new word for Logo

Now tell it to scream! (If it didn't work and you want to try again, you can't call it scream again unless you first enter `bye` to quit out of Logo and then start up Logo again. Instead, pick a new name like `scream1` or `scream2`.)

***　Enter your new command:**

```
scream
```

Logo runs the `scream` command.

Congratulations! You just wrote and ran your first program! A program is usually a group of instructions, and this had only one (`print`) but it's still a program. Soon you'll write longer, more complex programs that are just as easy to run.

You'll also write programs that do things with the turtle. This program didn't, but one of your next programs will have the turtle draw a whole flower when you enter the new command `flower` that you are going to teach to Logo.

> ***Try This!***
>
> Create a new program called `sing` that prints "la la la la la" on the screen.

Creating Programs with a Text Editor

What if you want to add more instructions to your `scream` program or change what its `print` command sends to the screen? You can't do this by entering `to scream` at the question mark prompt again; Logo will tell you "scream already defined." If you really want control over your program, you can write it with a text editor, which is what professional programmers do.

A ***text editor*** is a program that lets you edit ***text files***. What's a text file? It's a file that's not a binary file. What's a ***binary*** file? A binary (pronounced "bye-nurry") file is usually arranged in a special way so that only certain programs that know about that arrangement know how to read that file off of a computer's disk and do something with it. Pictures and music are almost always stored in binary files.

A text file is usually made up of letters, numbers, and symbols that you type on your keyboard. Text files can be read by all kinds of programs on all kinds of computers. Unlike most binary files, a text file from one computer can easily be moved to another computer and used there, even if the computer is running a different operating system.

When we write computer programs, we store them in text files, so to create and edit them you use a text editor program. There are many text editors out there, and many have special features that help programmers do their work more easily. Berkeley Logo comes with an editor called Jove. The name stands for Johnathan's Own Version of Emacs (Emacs is another text editor—my favorite one), and Jove is also another name for Jupiter, the god of the sky in ancient Roman mythology. On your computer, Berkeley Logo may be set up to use a different editor—probably the Notepad text editor if you're using Windows, something called "Logo Editor" on the Macintosh, or XEmacs (pronounced "ex-ee-max") if you're using Linux. Ask the person who set up Logo on your computer.

We're going to create a new command called `flower` in a text file with the name `flower.logo`.

* **The `editFile` command tells Logo to start up the text editing program. The part you put after `editFile` is the name of the file you want to edit. (Don't forget the double quote before the word so that Logo knows that it's the name of the file you want to edit and not the name of a special command to go with the `editFile` command.) The period (`.`) and the part after it in the filename is known as the extension. Not all files need an extension, but when you add ".logo" it shows that this file has a Logo program in it. Other programming languages have a similar habit: Java files usually end in ".java", C files in ".c", C++ files in ".cpp", and Perl program filenames in "pl". (As you can tell from the C++ and Perl examples, sometimes the filename extensions are abbreviated versions of the program language name.)**

Enter the following instruction at Logo's question mark prompt and execute it.

```
editFile "flower.logo
```

Logo starts up the editing program in its own window on your computer. (If the editing program asks you "Cannot find the flower.logo file. Do you want to create a new file?" pick "Yes" as your answer.)

* **Enter the following three lines. If you've ever used a text editor or word processor program to type in stories, reports, or e-mail, this should be easy. Use your Enter or Return key at the end of each line to start typing on a new line below the cursor's current line. Use your cursor keys to move the cursor up, down, left, or right if you need to fix a problem in something you already typed.**

The spaces at the beginning of of the second line won't change how the program works, but when you use spaces to indent the lines of a program in between the lines between the first and last lines, the spaces will make it easier to read the program. (OK, maybe it doesn't make this program easier to read, but it will when the programs get longer.)

```
to flower
  grow
end
```

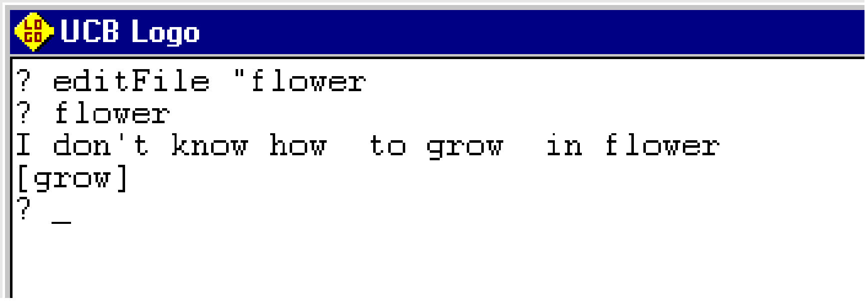
You should recognize most of what you've written: `to` tells Logo that you're starting a new program named after the next word ("flower"), end shows where the program ends, and the part in between (for this program, just one word) is the body of the program.

* **For now, you're finished typing the first version of your `flower` program. Save what you typed and quit out of the editor program. If you're not using a Macintosh, save your work by picking Save from the File menu and then pick Exit from the same menu to return to the Logo question mark prompt. On the Macintosh, picking Accept Editor Changes from the Edit Menu saves your work and returns you to the Logo question mark prompt.**

* **Enter and execute the following at the Logo question mark prompt to run your new program:**

```
flower
```

Logo executes the program, but doesn't know what to do with the `grow` part.



```
? editFile "flower
? flower
I don't know how  to grow  in flower
[grow]
? _
```

Logo finds an error in your program

* **Now you know what Logo does when it finds something in a program that it doesn't understand: it displays an error message about the line it doesn't understand so that you know what to go back and fix.**

Let's edit the program, take out the part that Logo doesn't understand, and replace it with something it does understand. Tell Logo again that you want to edit the `flower.logo` file.

```
editFile "flower.logo
```

* **Use your cursor keys to move your cursor after the word `grow` and then use the Backspace key (or, on a Macintosh, the delete key) to delete the letters in the word "grow."**

* **Add the three new lines shown in the darker, boldface text below between the `to flower` line and the end line of your program:**

```
to flower
  clearScreen
  setPenColor 6
  repeat 3 [forward 35 right 120]
end
```

This program has three instructions. The `clearScreen` command clears the screen and sends the turtle to the middle, as we've seen before. `setPenColor` here sets the drawing color to color number 6 for yellow. The third step draws a triangle by moving forward 35 turtle steps, turning to the right 120 degrees, and repeating these two steps for a total of three times.

Note that I didn't use abbreviations for any of the procedure names in this program. For example, I wrote `forward` instead of `fd`. This is because programs should be as easy to read as possible. If someone who had just learned Logo had to add something to a program that someone else wrote with a lot of

abbreviations in it, the programmer adding the new parts would have a harder time figuring out what it did.

* **Save your file again and exit out of the text editor program.**

* **Try your program again by entering the following at the Logo question mark prompt:**

```
flower
```

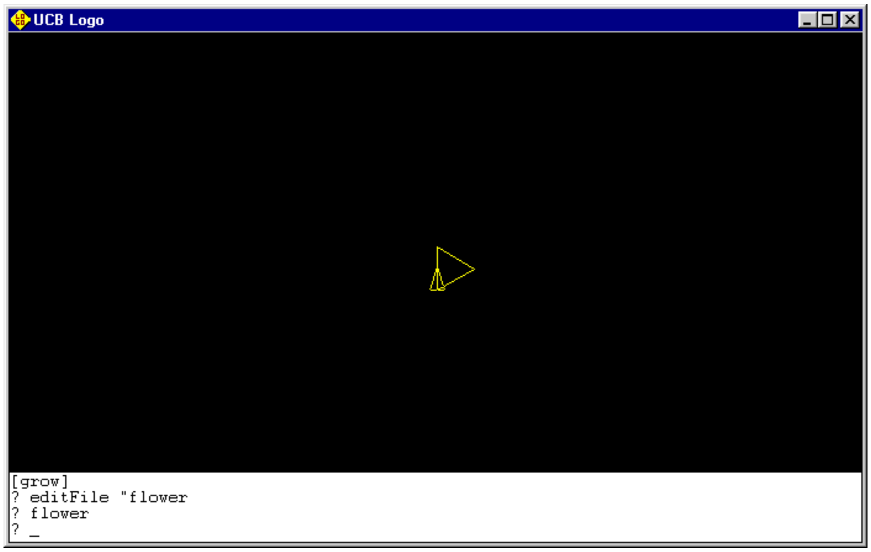
The screen clears, and the turtle draws a little yellow triangle.

* **If it doesn't draw the triangle you expected, edit the `flower.logo` program file again with the same instruction that you used before.**

```
editFile "flower.logo
```

Carefully compare your file with what you see in this book. See if you can find the problem and fix it, then save the file and try executing the program again.

Don't get frustrated if your program doesn't work correctly the first time and you have to go back and change one or two little things to make it run the way you want it to. This happens to the best programmers every day, and will happen to you as you type more programs in this book. Then you'll find your mistakes, which will be little, then fix them, and your programs will run great!



```
[grow]
? editFile "flower
? flower
? _
```

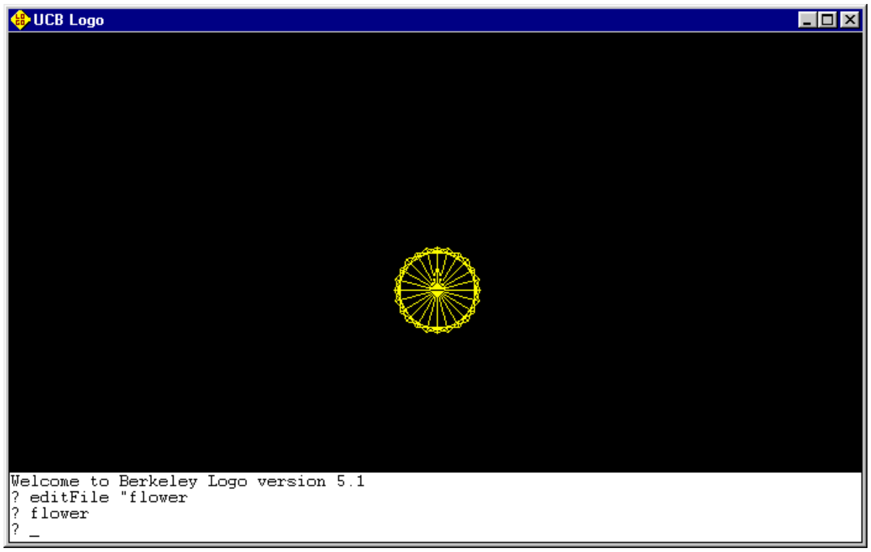
The flower program starts by drawing a yellow triangle

It's not much of a flower, but as with all programs, it's best to start simple, get it to work, and then build on what works. We're going to tell the turtle to make a flower by making a triangle, then turning to the right 15 degrees, then making another triangle, then turning right 15 more degrees, then making a third triangle, and so on. It will repeat these steps until the turtle goes all the way around in a complete circle, making a total of 24 triangles.

- * **Edit your `flower` program again and add the boldface text that you see below. Make sure that the fourth line has two left square braces (`[`) and two right square braces (`]`) as shown above. It will make a triangle like it did before, turn right 15 degrees, and repeat these two steps for a total of 24 triangles.**

```
to flower
  clearScreen
  setPenColor 6
  repeat 24 [ repeat 3 [forward 35 right 120] right 15]
end
```

- * **Save your program, exit out of the text editor, and execute your program the same way you did before.**



```
Welcome to Berkeley Logo version 5.1
? editFile "flower
? flower
? _
```

24 triangles in a circle make a flower

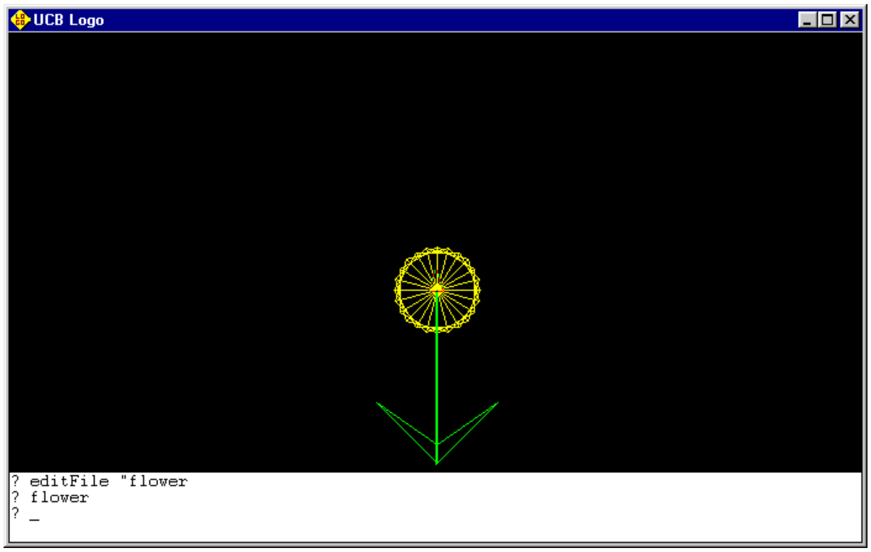
Now it looks a little more like a flower blossom. Let's add a stem and leaves under this blossom to make it look like a whole flower.

- * **Add the bold lines that you see below. The only new command is `back`, which is just like `forward`, except that it tells the turtle to move backwards. If the pen is down, it will draw just like it does when the turtle moves forward.**

```
to flower
  clearScreen
  setPenColor 6
  repeat 24 [ repeat 3 [forward 35 right 120] right 15]
  home
  setPenColor 2
  back 140
  left 45
  forward 70
  left 10
  back 60
  right 110
  forward 60
  left 10
  back 71
  left 45
  forward 140
end
```

Now, after the turtle makes the blossom part of the flower with 24 yellow triangles, it will go to its "home" in the middle of the screen, set the pen color to green (color number 2), and then do the lines and turns that draw the stem and leaves.

* **Compare what you typed with what you see printed in this book to make sure that you copied it exactly. Save your `flower` program, quit the editor, and run your program again. You should see a green and yellow flower appear. If not, edit the `flower.logo` file again and see which line or lines need to be changed.**



```
? editFile "flower
? flower
? _
```

The flower drawn by your program

Try This!

What would you change to have the turtle draw the blossom part in a different color? What would you change to make the blossom part bigger? (Hint: you want the turtle to draw bigger triangles.)

Making the Program Easier to Read

All this `left forward left back right forward` in the program makes sense to the turtle, but it's hard for people to read. It's important to make a program as easy as possible to read so that if someone needs to change it they'll know more easily which part they need to change.

There are two ways to make a program easier to read. All serious programming languages let you do both of these:

- *White space.* You indented the lines between the first line and the last line with a couple of spaces from your space bar. We call these spaces white space, because if you printed out the program with a regular printer there would be no printing there. You can also add white space by putting blank lines between the different sections of the program with your **Enter** key so that it's easier to see where each section starts and ends.
- *Comments.* Most programming languages give you a way to tell the computer "don't pay attention to what I wrote in this part here." This way, you can put a note in that part that explains what's going on to any person reading the program. That person could be you—all programmers have pulled out programs that they wrote a long time ago, tried to read them, and then thought "Why did I do this here in this part? What was I thinking?" That's when they see that if they had put more comments in, they would have an easier time understanding what they did.

In Logo, a semicolon character (`;`, pronounced "sem-ee-coe-lin") starts off comments.

***	Enter the following at the Logo question mark prompt and press Enter:**

```
; grow
```

Nothing happens. We saw earlier that Logo doesn't understand `grow`, but because of the semicolon at the start of the command that you just typed, Logo has no problem with it.

* **Enter and execute this instruction:**

```
forward 150   ; grow
```

The turtle moves ahead forward 150 turtle steps and that's it. Logo saw the instruction and the semicolon and knew that it should only execute everything before the semicolon. This means that you can put comments on the same line as an instruction that Logo should execute, as long as those comments come after a semicolon.

* **Edit your program to look like the one shown below. On the second line, put your own name and today's date instead of what you see here.**

```
; flower.logo
; Bob DuCharme October 1, 2002
; Draw a flower with the turtle.

to flower

  ; Set up the screen and turtle to get them ready.
  clearScreen
  setPenColor 6   ; Set the pen to draw yellow lines.

  ; Draw the blossom at the top of the flower.
  repeat 24 [ repeat 3 [forward 55 right 120] right 15]

  ; Draw the stem.
  home
  setPenColor 2   ; The rest of the drawing will be green.
  back 140

  ; Draw the left leaf.
  left 45
  forward 70
  left 10
  back 60

  ; Draw the right leaf.
  right 110
  forward 60
  left 10
  back 71
  left 45

  forward 140     ; One more line for the stem.

end
```

* **The first three lines are a good way to start off any program file that you create for any program that you write in any language: comments showing the name of the file, the program's author, the date he or she wrote it, and a summary of what the program does. If you always remember to do this, then you'll be more professional than a lot of professional programmers out there—too many of them forget this**

simple step.

Save your edited program, quit out of the text editor program, and run the flower program again to see if there are any problems. It should run the same way.

Source Code

In the last chapter, you loaded the `ttt` program and played a game of tic-tac-toe. Someone could load your `flower` program and run it if its program file was on their computer, as long as they had a Logo interpreter. If that person knew how to load `flower.logo`, they could run the `flower` procedure without needing to know `forward` or `setPenColor` or any of the other commands that you used when you wrote the program. You're the programmer, and they'd be using your program just like you've used other programs written by other programmers.

Try This!

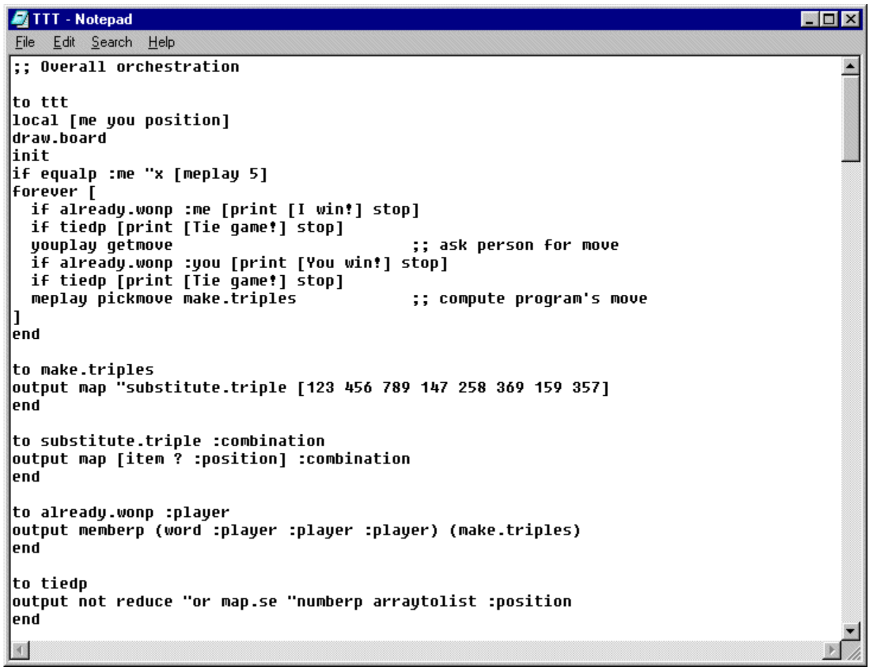
Have someone else come over to the computer, enter the instruction `flower` at the `?` prompt, and press **Enter**. Tell them that they're running the program that you wrote.

This person is your program's ***user***. When programmers talk about users, they're talking about the people they're creating the program for. Programmers are supposed to think a lot about their users and how their users expect the program to act. That's why a program designed for kids will look different from a program designed for adults, and why a program designed for musicians will look different from a program designed for business people.

Now that you're a programmer, you can look at other Logo programs and learn from them. Just as you edited your `flower` program by bringing it up in a text editing program, you can bring up programs written by other people in your text editor and take a look at them.

* **Use the `editFile` command to edit the `ttt` program. Remember that it's in a subdirectory, or folder, called `csls`. (Also, not that "ttt" is the whole name of this program in the `csls` folder—it doesn't have a ".logo" filename extension.)**

```
editFile "csls/ttt
```



```
;; Overall orchestration

to ttt
local [me you position]
draw.board
init
if equalp :me "x [meplay 5]
forever [
  if already.wonp :me [print [I win!] stop]
  if tiedp [print [Tie game!] stop]
  youplay getmove                         ;; ask person for move
  if already.wonp :you [print [You win!] stop]
  if tiedp [print [Tie game!] stop]
  meplay pickmove make.triples            ;; compute program's move
]
end

to make.triples
output map "substitute.triple [123 456 789 147 258 369 159 357]
end

to substitute.triple :combination
output map [item ? :position] :combination
end

to already.wonp :player
output memberp (word :player :player :player) (make.triples)
end

to tiedp
output not reduce "or map.se "numberp arraytolist :position
end
```

Looking at the ttt program with a text editor

* **The `ttt` program is long enough that it doesn't fit on the screen all at once. Use the Page Down key (which might be called PageDn or PgDn) to move down to more text in the file and PageUp to move back up.**

You can see that the program starts with the words `to ttt` just as your `flower` program starts with `to flower`. It also ends with the word `end`. It has more instructions after the `end` line, in groups that start with `to` and end with `end`. These procedures are like little helper programs that help the `ttt` program. (For example, the main `ttt` procedure has `make.triples` in it, and the second procedure begins with `to make.triples`.

* **See if you can find these words in the `ttt` program: `penup`, `forward`, and `penDown`. Do you see any square brackets? How about comments? Does it have indenting anywhere?**

* **When you're done looking at the `ttt` program, you can exit out of the editor without saving first because you're just looking at it, not editing it. If you accidentally typed in even a single character somewhere, your editing program might ask if you're sure that you want to quit or if you want to save your changes first. You are sure that you want to quit, and you don't want to save your accidental changes.**

You used the text editor to look at the ***source code*** of the `ttt` program. The source code is the file full of text that someone typed out as instructions to the computer when they wanted to write a program for people to run. That's why, when you spend some time writing a program's instructions, you can tell people that you were "coding" and you'll be talking like a real programmer.

> ***Try This!***
>
> The `csls` folder also has programs named `plot`, `poker`, and `tower`. Take a look at them with your text editor the same way you looked at `ttt`.

Variables

Imagine that you have an envelope with the word "FLAVOR" written on it. One day, you write the word "CHOCOLATE" on a little piece of paper and put it inside the envelope. You walk into an ice cream store and say "I would like an ice cream cone, please." The lady behind the counter asks "what kind of ice cream?" and you hand her the envelope. She sees the word "FLAVOR" on the front, looks inside the envelope, takes out the paper, and sees that it says "CHOCOLATE." She gives you the envelope back, and makes you a chocolate ice cream cone.

The next day, you write the word "VANILLA" on another piece of paper and put it in the same envelope. You go into the same store and the lady behind the counter says "oh great, it's the kid with the envelope." While you hand it to her, you say "ice cream cone, please!" She looks inside, reads the new piece of paper, gives you back the envelope, and makes you a vanilla ice cream cone.

The envelope is a container for some information, and this container has a name. The name of this envelope, "FLAVOR," describes the information inside. (If the envelope had said "ZX30L2" on the front when you handed it to the lady behind the counter, she probably would have said "Hey, what's your problem! Take your envelope and get out of my store!") The information inside the container can change; one day the FLAVOR container stores the information "CHOCOLATE" and another day it stores the information "VANILLA."

All programming languages have something called ***variables*** (pronounced "vair-ee-uh-bulls") that are containers for information. Programs can pass them around from one instruction to another, just like you handed the envelope to the ice cream lady. Programs can look inside the variables to see what information they store, and they can change the information inside. It's very common for computer programs to spend much of their time looking inside of variables, checking the information, and then doing different things depending on what they found there. It's very similar to what the ice cream lady did when she looked inside the FLAVOR envelope and then gave you one ice cream flavor or another depending on what she found there.

In Logo, the command to make a variable is `make`. (In other programming languages, creating a new variable is usually called ***declaring*** a variable.)

* **Enter the following instruction to make a variable named `flavor`. Don't forget the double quote char-**

acter before the variable name `flavor` and before the value that you're putting into that variable.

```
make "flavor "chocolate
```

In other programming languages, just like in English, a double quote goes at the beginning and end of the word or words being quoted, "like this" (or like this: "chocolate"). In Logo, you put one before each word that you want to treat like a little piece of *data*, or information for the computer, but not after. Remember to put each piece of data right after the double quote, or Logo will give you an error message. Watch out for this when you type in your programs and get errors when you try to run them, because this is a common source of errors.

* **Tell Logo to print the information stored in the `flavor` variable. The `thing` operation is a way of saying "the thing called flavor."**

```
print thing "flavor
```

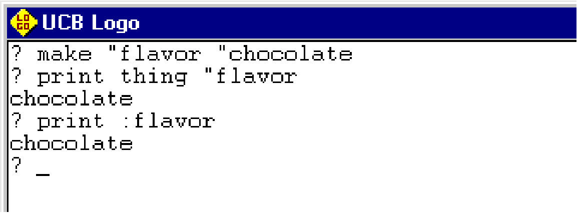
Logo tells you what information is stored inside the `flavor` container—what we call the *value* of the variable.

* **As you learn various programming languages and decide what you like and don't like about each, you'll sometimes find yourself thinking "whoever designed this language sure picked a great name for that" or "they sure picked a stupid name there!" The name `thing` may inspire one of these responses.**

Luckily, because programs check the values of variables so often, Logo provides a shortcut: the colon (`:`) character. It's located on the key next to your `L` key, and you'll have to press the Shift key with it if you don't want to type the semicolon (`;`) character instead. Try this instruction at your question mark. It should give you the same result as the previous instruction:

```
print :flavor
```

The colon is a replacement for the word "thing" and the double quote that follows it. It's so much easier to type the colon that we won't use the word "thing" again for checking variables.



```
? make "flavor "chocolate
? print thing "flavor
chocolate
? print :flavor
chocolate
? _
```

Printing out the value of the flavor variable

* **Now change the value of the variable with this instruction:**

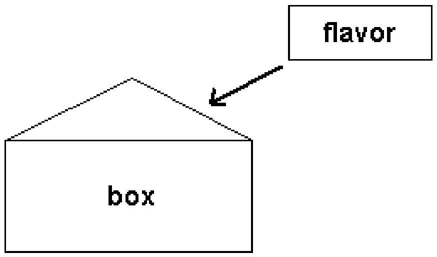
```
make "flavor "vanilla
```

* **Use the same `print` instruction that you used before to find out the variable's value.**

* **Try these two instructions. See if you can guess what will happen before you execute the second one.**

```
make "box "flavor
print :box
```

This is kind of like putting a piece of paper that says "flavor" on it (not an envelope that says "flavor" on it) into an envelope that says "box" on it. The `box` variable doesn't know that you had created another variable named `flavor`; it just knows that you put the value "flavor" into the `box` variable.

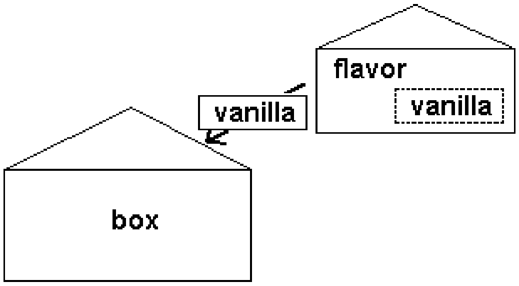


Storing the value "flavor" in the box envelope

* **Now try these two instructions. They're the same as the last two, except that you're not putting the double quote before the name `flavor`.**

```
make "box :flavor
print :box
```

This time, you're telling Logo to put the value of the `flavor` variable inside of the `box` variable, which is why `box` now has the value "vanilla" in it.

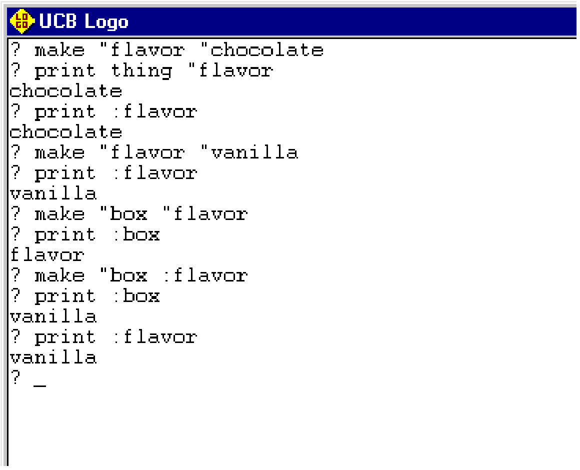


Storing a copy of the value in flavor ("vanilla") in the box envelope

* **Now that `box` has the value "vanilla," what is the value of the `flavor` variable?**

```
print :flavor
```

It still has the value "vanilla." When you created the `box` variable, you told Logo to put a copy of the `flavor` variable in it. It did, and it didn't change the value of the `flavor` variable.



```
? make "flavor "chocolate
? print thing "flavor
chocolate
? print :flavor
chocolate
? make "flavor "vanilla
? print :flavor
vanilla
? make "box "flavor
? print :box
flavor
? make "box :flavor
? print :box
vanilla
? print :flavor
vanilla
? _
```

Copying variable values to other variables

Don't worry too much about copying variables to other variables. This was just an example of the more interesting things that you can do with variables.

* **Variables can also store numbers, and you don't need to put the double quote character before the number value. Try the following instruction to create a variable called `green`, and then check the variable's value with the `print` command.**

```
make "green 2
```

* **What if you print the value of a variable that doesn't exist? You never created a `color` variable, so try this:**

```
print :color
```

You never assigned any value to a `color` variable, which is what Logo tells you.

> ***Try This!***
>
> Try setting the `flavor` variable to other values and then checking to make sure that the values you put there are really there.
>
> Can you make up a new variable different from `flavor` and then change its value a few times? (Make sure that it has no spaces in its name.)

There's one more thing about the `thing` thing: once we learned it, we learned about using the colon (`:`) abbreviation so that we wouldn't have type out the procedure name "thing." Unlike other procedure names that we've learned such as `forward`, `print`, and `make`, `thing` is not a command. If the `flavor` attribute had the value "vanilla" and you typed this at the question mark prompt,

```
thing "flavor
```

Logo would tell you "You don't say what to do with vanilla." Because `thing` is an ***operation***, and not a command, it outputs a value to get used by a command or by another operation. If you type the instruction above, Logo does know that it should look in the `flavor` variable and pull out its value, but then it wants to output it to a command that will do something with it, and this instruction doesn't have one. That's why it says "You don't say what to do with vanilla." (In other programming languages, instead of calling this an operation that outputs a value, we usually call it a function that returns a value.) If you enter this,

```
print thing "flavor
```

you are giving `thing` a command to output to: `print`. The `print` command knows what to do with it; it puts it on the screen. The important thing to remember is that of the various special words you learn to use in Logo, some are commands and some are operations. Commands tell Logo to do something, and operations output a value to be used by something else—usually by a command, but sometimes by another operation. (If it's used by another operation, there has to be a command somewhere in the instruction using the result of all this outputting.) In the next chapter, we'll learn about some very handy operations to combine with the commands that you've already learned.

Procedures? Commands? Operations?

Nearly all of the special words that mean something in Logo are procedure ("pro-seed-jer") names. There are two kinds of procedures: ***commands*** and ***operations***. Logo comes with its own built-in procedures (for example, commands such as `forward` and `print` and operations such as `thing` and more that we'll learn shortly) and you can write your own new ones. You've already written two command procedures: `scream` and `flower`.

Let's edit your `flower` program to use variables.

* **Bring up your `flower` program in the text editor.**

```
editFile "flower.logo
```

* **Add the four lines that you see in bold below that begin with the word "make" to your program, along with the comment line above them. For the two `setPenColor` lines that you wrote earlier, delete the number after each command and put the name of the color variable (along with the colon before it) there instead.**

```
; flower.logo
; (your name here) (today's date here)
; Draw a flower with the turtle.

to flower

  ; Set color values.
  make "red 4
  make "blue 1
  make "yellow 6
  make "green 2

  ; Set up the screen and turtle to get them ready.
  clearScreen
  setPenColor :yellow   ; Set the pen to draw yellow lines.

  ; Draw the blossom at the top of the flower.
  repeat 24 [ repeat 3 [forward 55 right 120] right 15]

  ; Draw the stem.
  home
  setPenColor :green   ; The rest of the drawing will be green.
```

```
    back 140

    ; Draw the left leaf.
    left 45
    forward 70
    left 10
    back 60

    ; Draw the right leaf.
    right 110
    forward 60
    left 10
    back 71
    left 45

    forward 140     ; One more line for the stem.

  end
```

* **Save your new version of the `flower` program, quit out of the text editor, and run your flower program. It should run exactly as it did before.**

* **Bring up your `flower` program in the text editor again and change the line that says**

```
setPenColor :yellow
```

to say this:

```
setPenColor :red
```

* **Save the program, exit out of the text editor, and run the program to see what happens.**

See how variables can make a program easier to read? In fact, once you changed the line that said this

```
setPenColor 6   ; Set the pen to draw yellow lines.
```

to this,

```
setPenColor :yellow   ; Set the pen to draw yellow lines.
```

you don't even need the comment that says "Set the pen to draw yellow lines" anymore.

The program is not only easier to read, but easier to write. For example, imagine that you were going to have your turtle draw a sun above your flower. When you can write `setPenColor yellow` in your program, you don't have to remember which number was the number for yellow.

Variables do much more than make programs easier to read and write. They make a program more flexible, because they let a single program do a lot more different things. (That's why we played a little with copying a value from one value to another.) In the next chapter, we'll write a program that uses variables to create a math game for your computer. If you don't like math, remember: you can make the game as easy or as hard as you want.

What We Learned

In this chapter, we learned:

- How to create loops, which tell Logo to repeat some instructions more than once.
- How to write a program and run it.
- How to use a text editor to create new programs and to edit existing programs.
- Why white space at the beginning of some lines of your program and between other lines can make a program easier to read and understand.
- What comments are and why they make a program easier to understand.
- How to make words appear on the screen with the `print` command.
- How to look at other Logo programs that come with Berkeley Logo.
- How to create variables, how to put information into them, how to check their values, and how to copy their values to other valuables.

New Commands and Operations in This Chapter

(See the "Procedures Reference" Appendix in the back of the book for brief descriptions of what these do.)

New Commands

- `back`
- `editFile`
- `end`
- `hideTurtle`
- `make`
- `print`
- `repeat`

- `showTurtle`
- `to`

New Operations

- `thing`

More Things to Try

1. Use your text editor to write a program called `bigBlueSquare` that clears the screen and then draws a big blue square. If you find yourself typing the same instruction over and over in your program, remember: computers are there to do repetitive jobs for you! Use a loop instead.

   Also remember to include comments in your `bigBlueSquare` program.

2. After you write and run `bigBlueSquare`, edit it to add two new instructions and then run it again:

   - One instruction at the beginning creates a variable called `message` with the value "what a great square!"
   - One just before the `end` instruction that displays the value of the `message` variable with the `print` instruction after the turtle draws the square.

   Change the value of the `message` variable to something else and see if that message shows up when you run the `bigBlueSquare` program.

Creating a Math Game

Have you ever played a computer game that asks you math questions and gives you a higher and higher score when you answer more and more questions correctly? In this chapter, you're going to write a program called `mathGame` that does just that. You can make the math questions as hard or as easy as you want. As you put this program together, you'll learn programming tricks that will help in all kinds of programs that you write, even those that have nothing to do with games or math.

Telling Logo to Do Math

To have Logo do math, you don't need any new commands or operations. Just put a math expression after a `print` command and Logo will do the math and put the result on your screen.

* **Let's start with some simple math. Enter the following and press Enter. Don't use any square braces like you did with the `print` command before. You'll find a plus (+) sign on the same key as the equals sign (=). Press and hold down the Shift key when you type a plus sign so that you don't accidentally type an equals sign instead.**

```
print 3+2
```

What does Logo print?

* **Let's try adding some bigger numbers. Enter and execute this:**

```
print 3421 + 2234
```

Logo adds them as easily as it added 3 and 2. It also didn't care whether you put a space on either side of the plus sign or not. (Some Logo interpreters do care about this.)

* **How about adding more than two numbers? Try this:**

```
print 5+3+2
```

* **What about really big numbers? Try entering this. (Each of the two numbers has 15 nines.)**

```
print 999999999999999 + 999999999999999
```

The answer may look a little odd: `2e+015`. This is known as *scientific notation*. Scientists and mathematicians use this when they want to write very big or very small numbers.

Try This!

> Add more combinations of numbers together—big numbers, little numbers, two numbers, lots of numbers, two numbers, zeros, or anything you like.

* **Let's try subtraction. Enter and execute this:**

```
print 53 - 20
```

* **For multiplication, most computer languages don't use the little × sign that you probably write for multiplication at school—there's no key on your keyboard for it, so how would you type it? Instead, use the asterisk (`*`). This is on the same key as the number 8. Don't forget to press and hold down the Shift key when you type it. This way, you'll really type an asterisk and not an 8. Try this instruction:**

```
print 2 * 5
```

* **Try multiplying more than two numbers:**

```
print 2 * 3 * 4
```

* **To divide two numbers, use the slash key. Ask Logo to 12 divided by 4:**

```
print 12/4
```

* **If you ask Logo to divide something that doesn't divide evenly, it uses the dot known as a decimal point to show the result as a decimal number. Try dividing 10 by 4:**

```
print 10/4
```

The answer tells you that 4 goes into 10 two and a half times. (Two fours make eight, and half of another four added to this makes ten.)

* **How about combining multiplication and addition? Try this:**

```
print 2 + 3 * 4
```

What was the answer? Did it add 2 plus 3 and then multiply the answer by 4, or did it multiply 3 times 4 and then add 2 to the answer?

* **Let's try the last one again, but change the order:**

```
3 * 4 + 2
```

What was the answer? Did it perform the multiplication or the addition first?

You probably figured out that both times it multiplied before it added. This isn't a special Logo thing; it's a math rule about how you combine addition and multiplication. Logo is just following those rules.

Many people who hate math like it more when they're using computers. Why? Because the computer takes care of the boring parts and they can think more about the interesting parts, like the different patterns that numbers make when you do different things with them. Just as you used numbers to draw the picture of the flower in the last chapter, you can use math and drawing instructions together to see a picture of just how interesting some of those patterns can be.

- **We can draw an interesting pattern with a one-line instruction using the `repCount` operation. First, to see what `repCount` does, try this:**

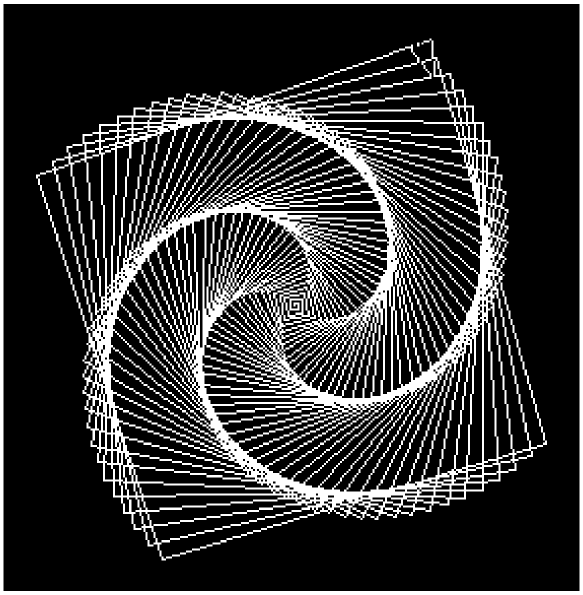
  ```
  repeat 5 [print repCount]
  ```

  It outputs a number showing the count of the repetition. When you execute this instruction you'll see that the first time the `print` command is executed, `repCount` has a 1 in it, the next time it has a 2 in it, and so on until it has a 5 in it at the end.

- **Now we'll tell the turtle to move 200 times, and each time we'll have it move the distance output by the `repCount` operation (first 1, then 2, then 3, and so forth). After each move, it will turn right 89 degrees. Execute this instruction:**

  ```
  repeat 200 [forward repCount right 89]
  ```

  You only typed one line, and look what a complex picture it makes!



Using math and repCount to draw a picture

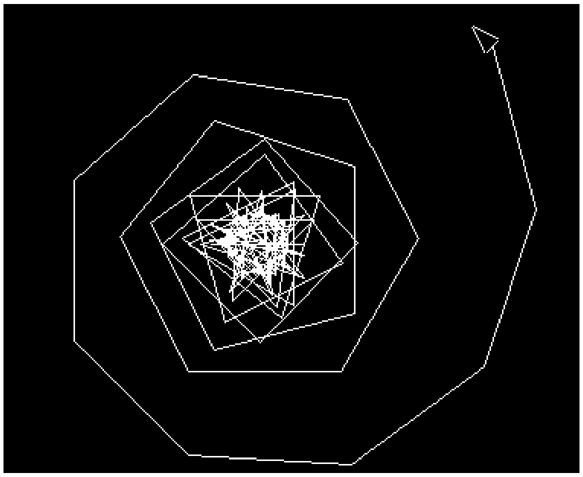
* **Our next picture will use multiplication. Clear the screen to give the turtle a blank screen before drawing it.**

```
clearScreen
```

* **The next instruction uses multiplication to change the amount of turning with each step. By multiplying repCount by 3, the turtle will turn 3 degrees the first time, 6 the next time, 9 the next time, and so forth until it turns 330 degrees on the 110th step:**

```
repeat 110 [forward repCount right repCount * 3]
```

It's another strange picture:



Using repCount with multiplication to draw a picture

Sometimes multiplication can be fun, especially if you let the computer do the multiplying!

Try This!

Changing any of the numbers in that last instruction can make a completely different picture. For example, try clearing the screen with the `clearScreen` command and entering this:

```
repeat 100 [forward 20 right repCount * 5]
```

Try changing the numbers to anything you like. You don't have to multiply `repCount`; you can divide with it, use addition or subtraction with it, or leave it alone and play with the other numbers. If you want, store this instruction in a program file and run it. If you put the `setPenColor` instruction in there, the pictures could be even more interesting. Use the `clearScreen` command between each picture to make room for each new picture.

Mathematicians often use computers to draw pictures that use the math that they're working with just like you used multiplication to draw the picture above. For more complex math, pictures can make it easier to understand the effect of some math that's being done to a group of numbers.

* **What if you did put square braces around your math expression when you print it? Try the first math expression again with square braces around the numbers:**

```
print [3+2]
```

Is the result different? With the square braces, Logo treated the thing to print as a string of characters.

Numbers or Not?

Sometimes computer programs have numbers that we don't want to treat as numbers. For example, you may think of your phone number as a number, but unlike the price of a candy bar, no one is ever going to do math with a phone number. If you want to buy a candy bar and a pencil together, you add together the price of the candy bar and the price of the pencil to find out what how much money you must pay. No one will ever add a number to your phone number to figure something out. It's not a quantity, or an amount of something.

We call a number such as a phone number that doesn't represent a quantity a ***string*** of characters, or just a string. Other kinds of character strings are just about anything with words and letters in them—your name, the title of a song, or even the words in this book. Can you think of any other numbers that don't represent quantities and get treated as strings? (Hint: think of the parts of your mailing address.)

In programming languages that make you declare variables at the beginning of programs, you have to say whether they will be used for strings, numbers, or other types of data. Logo is more flexible about this, so you don't have to worry about the type of data stored in each value.

Try This!

Try more combinations of addition, multiplication, subtraction and division at the question mark prompt.

Creating Interactive Programs: Having Your User Set Variable Values

An ***interactive*** program is one that the user can interact with. In other words, the user can change what the program does by clicking the mouse, typing in instructions, picking things off of menus, or even by talking into a microphone or turning a steering wheel attached to the computer. Almost every program you've ever used on a computer was interactive.

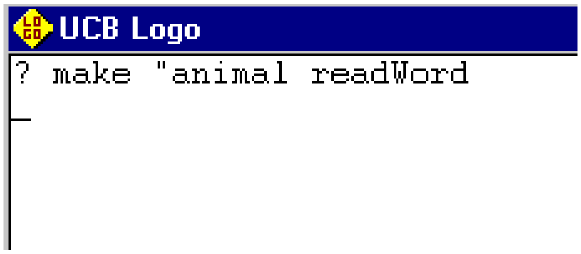
There is one program that you've run that wasn't interactive: your `flower` program. When someone ran it, whether it was you or another friend, the turtle drew the flower and then the program was done. The program never waited for the user to do anything when you told it to start running; it ran from its first instruction to its last one without stopping as soon as you told it to. (How about the tic-tac-toe program that you ran—was that an interactive program? Why or why not?)

When you click an icon with your mouse or turn a steering wheel while playing a car racing game or tell a tic-tac-toe game where you want your X, the program is probably taking information about what you did and storing it in a variable. Another part of the program then checks the variable to figure out what to do, just as the lady in the ice cream shop in the last chapter checked the envelope you gave her to see what kind of ice cream to give you. You can make your programs interactive using commands like `readWord`, which gets information from the user and stores it in a variable. Then, your program can check the variable's value and do different things depending on what it finds.

* **Enter the following instruction and press Enter:**

```
make "animal readWord
```

It looks like nothing happened:



```
? make "animal readWord
_
```

Trying out the readWord operation

* **Now enter the word "dog" and press Enter again:**

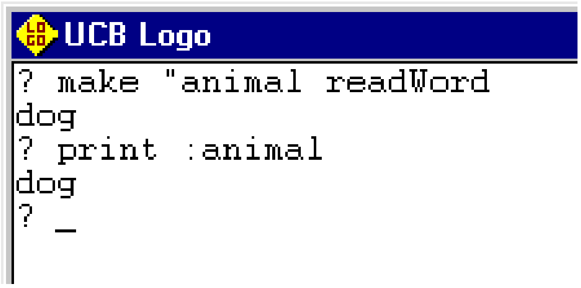
```
dog
```

The question mark reappears.

* **Ask Logo about the value of the `animal` variable:**

```
print :animal
```

It was set to "dog"!



```
? make "animal readWord
dog
? print :animal
dog
? _
```

readWord put a value in the animal variable

Let's review what happened here.

If you entered this instruction,

```
make "animal "dog
```

you would be setting the `animal` variable to have the value "dog." But you didn't do this. Instead of a value with a double quote in front of it, you put the name of a Logo operation:

```
make "animal readWord
```

The `readWord` operation tells Logo to wait for the user to type some text and to then grab all the text that the user types until he or she presses the **Enter** key. The name `readWord` doesn't describe this operation very well, because it can read more than one word: as many as you want to type before you press **Enter**.

When you use `readWord` with the `make` command like this, Logo stores the grabbed text in the variable whose name you put after the `make` command. That's why your user input (the part that you typed in when `readWord` waited for you) got stored in the `animal` variable.

Try This!

Try the same instruction a few more times, entering other values besides "dog," and then check the `animal` variable's value with the `print` command each time. Try entering numbers, or a couple of words. What happens if you don't enter anything when `readWord` is waiting for your input, and you just press the **Enter** key without typing in something first?

How will our math game program use variables?

When you add numbers together, the numbers that you're adding are called the addends, and the result of adding them together is the sum. (These are math words, not Logo words.) For example, if I say that 2+3+4=9, then 2 and 3 and 4 are the addends and 9 is the sum.

Our math game will add two numbers together after storing them in variables. We'll call the two variables `addend1` and `addend2`. The "1" (the number one, not a lower-case "L") and "2" in the names have nothing to do with the values stored in these variables; they're just there to give the two variables different names. You could still store 1 or 2 or 0 or 1000 or anything you want in either variable.

Before we write a program to add these numbers, let's play with `addend1` and `addend2` at the question mark prompt to get used to doing math with variables.

* **Create your `addend1` variable with a value of 3:**

```
make "addend1 3
```

Use the `print` command to make sure that the 3 got stored in the `addend1` variable.

* **Create your `addend2` variable with a value of 4:**

```
make "addend2 4
```

Use the `print` command to check on this one as well.

* **Have Logo do some math with the two variables:**

```
print :addend1 + :addend2
```

What did Logo do?

Your First Interactive Program: An Adder

Before we create our `mathGame` program, let's create a simpler interactive program to add two numbers. We'll call it `adder`. After asking the user for two addends, it will add the two together and print the answer on the screen.

* **Start up the editor and tell it to create a Logo program file called `adder.logo`:**

```
editFile "adder.logo
```

* **Type the following lines into your text editor to create your `adder` procedure:**

```
; adder.logo
; (your name here) (today's date here)
; Add two numbers input by user.

to adder

  ; Get the input from the user.
  print [Enter the first addend]
  make "addend1 readWord

  print [Enter the second addend]
  make "addend2 readWord

  ; Tell the user the result.
  print [The sum is]
  print :addend1 + :addend2

end
```

* **Save the program and exit the text editor to return to the Logo question mark prompt.**

Before running your new program, let's look at what it does. Between the `to adder` line and the `end` line, it has three parts:

1. The first part has two lines after a comment line that describes what those two lines do. If the `adder` program just started off with the line that has the `readWord` operation, the program's user would start the program and then just see a blank line. Remember the first time you tried the `readWord` operation by entering `make "animal readWord`? You pressed the **Enter** key and it looked like nothing happened. You only knew that you should enter the word "dog" under the `readWord` instruction because this book told you to. In our `adder` program, the user needs to know when the `readWord` operation is waiting for them to enter something. This is why the `adder` program has a `print` command to tell the user what to enter ("Enter the first addend") before the line with the `readWord` operation gets the value from the user. Like the question mark where you type instructions to the Logo interpreter, the phrase "Enter the first addend" is called a prompt.

2. The second part of the program is a lot like the first, except that the prompt tells the user to enter the second addend and the program stores the value that the user enters in the `addend2` variable.

3. The third part of the program prints the phrase "The sum is" and then prints the sum of `addend1` and `addend2`.

Try This!

Run your `adder` program. Run it many times. Try entering two small numbers for the addends, two big numbers, two zeros, and any combinations of these you can think of. Try telling it to add "cat" and "dog." What does it do? Later in this chapter we'll learn how to make your programs handle bad input like that better.

It would be nice if the `adder` program printed the words "The sum is" and the answer all on the same line. Let's try combining these words and the variable values on one line of the screen.

* **Run `adder` one more time, and give it two regular numbers as input to make sure that it doesn't have strange values like "cat" or "dog" left over from your testing earlier.**

* **Execute the following instruction at the question mark prompt:**

  ```
  print [The sum is :addend1 + :addend2]
  ```

  Logo prints exactly what you told it to on the screen: "The sum is :addend1 + :addend2". When you put a list of words inside of square brackets after a `print` command, Logo prints exactly what it found between those brackets on the screen.

  We don't want the names of the variables (`addend1` and `addend2`) to show up after the words "the sum is." We want their values to show up there.

  Instead of having `print` put a list of text expressions on the screen, we'll have the `sentence` operation combine the text and numbers we want and then have the `print` expression put the result on the

screen.

* **First let's store the result of adding the two addends in its own variable called `total` with this instruction:**

```
make "total :addend1 + :addend2
```

* **The `sentence` operation combines everything you give it into one list. You can give it strings of text that you want to show up exactly as they are by putting a double quote before each one. Anything without this double quote must be something that Logo can understand, such as a number or a variable name with a colon. Enter this instruction:**

```
print (sentence "The "sum "is :total)
```

When you enter this, Logo prints the same sentence as your `adder` program did, but with the sum on the same line as the words that introduce it.

In the last chapter we learned that `thing` is an operation, not a command, and that operations output information to be used by a command (or by another operation!) In the instruction that you just typed, `sentence` is also an operation, which outputs its information to the `print` command in the instruction above.

Try This!

Edit your `adder.logo` program to print one line instead of two at the end of the program. For example, if the addends are 4 and 5, the output should say "The sum is 9".

Being able to combine regular strings of text with the values of variables will be important when we get to the math game.

Picking Random Numbers

If the math game will pick two numbers and ask the game player to add them, how will it pick these two numbers? With my favorite part of any programming language: the `random` operation. (In other languages, it's usually called the random function.)

* **If your computer is using the Windows operating system and there are any graphics on the screen, enter the `clearText` command to clear the graphics and set up your whole screen for text output:**

```
clearText
```

* **Execute the following instruction 10 times. You can type it in and execute it 10 separate times, but it would be faster and easier to use a `repeat` loop to make it happen 10 times.**

```
print random 4
```

It didn't always produce the same result. What was the highest number you saw? What was the lowest?

* **Try the same thing again, only with the number 12 as input to the `random` operation. Do this 10 times as well.**

```
print random 12
```

What was the highest number you saw? What was the lowest? Try it again to see if you get the same numbers.

The random operation looks at the number that you give it as input and picks a number less than that to output. If the number is 4, the number it picks could be anything from 0 to 3; if the number is 1000, it will pick a number from 0 to 999.

What do you do with these numbers? Anything you want! Think of a place where you would use a number, and then use a random number instead. For example, when you draw with your turtle, three commands that need a number to do their job are `setPenColor`, `forward`, and `right`. Using the `random` operation with these commands, you can write a short little program to draw crazy pictures, and it draws a different crazy picture every time you run it.

* **Create a program file called `crazyPic.logo`:**

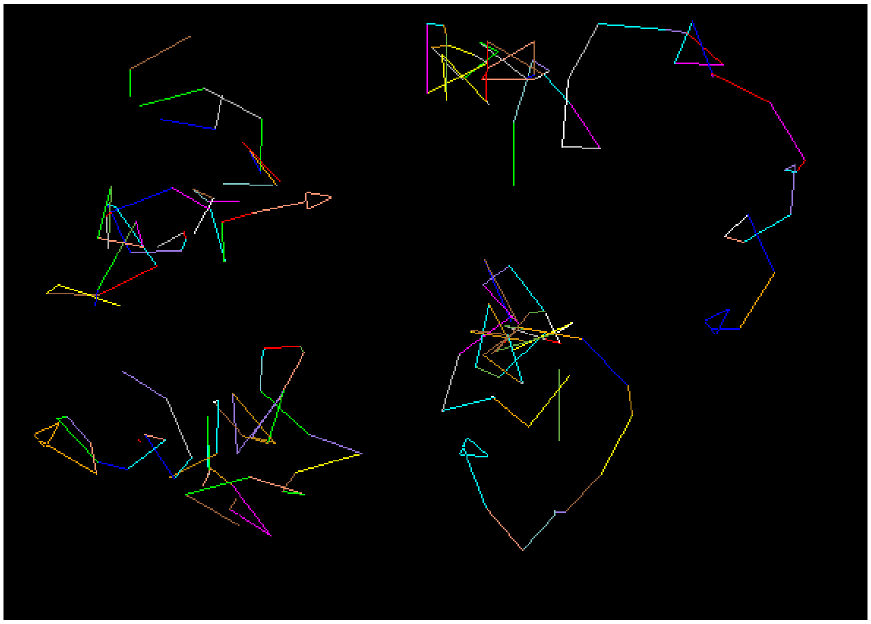
```
editFile "crazyPic.logo
```

* **Type in the following text to be your program:**

```
; crazyPic.logo
; (your name here) (today's date here)
; Draw a crazy picture!

to crazyPic

  clearScreen

  repeat 50 [
    setPenColor random 16  ; pick a color at random
    forward random 50      ; move forward a random amount
    right random 180       ; turn right a random amount
  ]

end
```

* **Save it, quit the editor, and run your `crazyPic` program.**

* **Then run it again, and again and again! The picture is different every time.**



Four different pictures created by running the same crazyPic program four times

* **Try changing some of the numbers in the program and see what happens. For example, what if it only picks from four different random colors instead of 16? What happens if the turtle can move or turn more or less than the 180 degrees given in the program above? If it makes more than 50 or less than 50 lines to make each picture, how does the picture look different?**

Drawing pictures is just the start of the fun you can have with the random operation. For example, some programming languages and Logo interpreters give you a way to play musical notes on the computer's speaker. Usually, you use numbers to tell the computer which notes to play. When you can do this, the random function lets you create a program that makes up music just as the little program above makes up pictures.

Having Your Program Make Decisions

What if Ringo the Robot could look inside of an envelope called "flavor" and then give you the flavor of ice cream that was written on the piece of paper inside? If Ringo really existed, it's more likely that instead of looking inside of an envelope he would check the value of a variable called `flavor` instead. Most robots have computers attached to them, and looking at variables is much easier for a computer than reading words off of a piece of paper.

No matter how Ringo found out what flavor you wanted, he would have a decision to make. Sometimes he would have to give you chocolate ice cream, and other times he would have to give you vanilla ice cream. How can you tell

Ringo, or a computer, to do one thing if a variable has one value and another thing if the variable has another value?

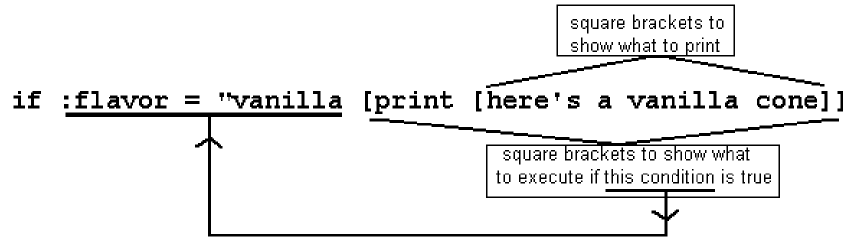
Logo's `if` command lets you do this. This command gets two inputs: first you describe a condition that will be either true or false, and then inside of square brackets you list one or more instructions to perform if the condition is true. Let's try it.

* **At Logo's question mark prompt, create a variable called `flavor` with a value of "chocolate."**

```
make "flavor "chocolate
```

* **Enter an `if` instruction that tells Logo "if the value of the `flavor` variable equals 'chocolate' then print the message 'here's a chocolate cone.' " (Notice how it has two pairs of square brackets: one around the part to "print" on the screen, just like we've always done, and another around the whole `print` instruction. That's the one that shows the `if` command what to execute if its condition is true.)**

```
if :flavor = "chocolate [print [here's a chocolate cone]]
```


square brackets to show what to print
if :flavor = "vanilla [print [here's a vanilla cone]]
square brackets to show what to execute if this condition is true

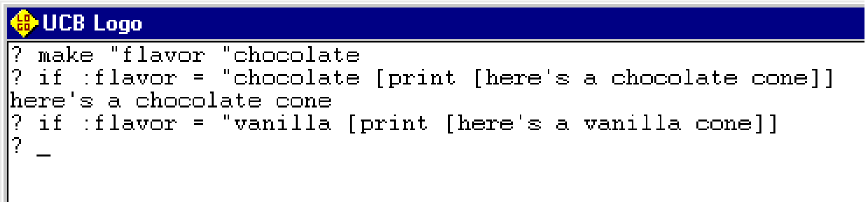
Which square brackets do what

The message prints, because the `flavor` variable does have a value of "chocolate."

* **Try this similar instruction:**

```
if :flavor = "vanilla [print [here's a vanilla cone]]
```

When you execute the instruction, it looks like nothing happens. Why? Because you told Logo "if the value of the `flavor` variable equals 'vanilla' then print the message 'here's a vanilla cone.' " The value of the `flavor` variable is still "chocolate," and the condition that the `if` instruction checks is false, so it doesn't print its message.



```
? make "flavor "chocolate
? if :flavor = "chocolate [print [here's a chocolate cone]]
here's a chocolate cone
? if :flavor = "vanilla [print [here's a vanilla cone]]
? _
```

Using the if command to check the flavor variable's value

If a real ice cream store only had chocolate ice cream, and you asked for vanilla, wouldn't it be a little strange if they didn't do or say anything at all? Wouldn't it make more sense for them to tell you, "Sorry, we only have chocolate"? If you want to perform one action (or a list of actions) when a condition is true and another when the condition is false, you can do this in Logo with the `ifElse` command.

This command is like `if`, but with a little more. Like `if`, `ifElse` starts with an expression that is either true or false. Then, inside of square brackets, it has one or more instructions to execute if the expression is true. The `ifElse` command also has another pair of square brackets after that, with the instructions to execute if the expression isn't true.

An `ifElse` command can get pretty long, so instead of typing the whole thing at Logo's question mark prompt, let's store one in a program. Then we can run it over and over and make changes if we want.

- **Create a program file called `iceCream.logo` by entering the following instruction at the question mark prompt:**

  ```
  editFile "iceCream.logo
  ```

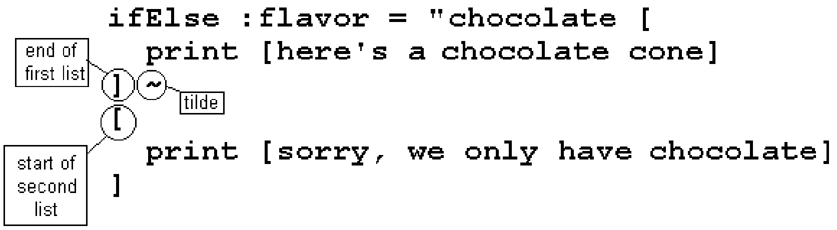
- **Type the following lines into your editor to create the `iceCream.logo` file:**

```
; iceCream.logo
; (your name here) (today's date here)
; Test the ifElse command.

to iceCream

  ; Get input from user, store in flavor variable.
  print [What kind of ice cream do you want?]
  make "flavor readWord

  ifElse :flavor = "chocolate [
    print [here's a chocolate cone]
  ] ~
  [
    print [sorry, we only have chocolate]
  ]
```

```
end
```

Most parts of this program look just like things you've typed into other programs before: it starts with a comment describing what the first part does, then prints a prompt asking the user a question, then reads the user's input into the `flavor` variable. The `ifElse` statement starts off with the condition `:flavor = "chocolate` and follows it with a list of instructions to execute if the condition is true. (This "list" is really just one instruction: `print [here's a chocolate cone]`.) Another pair of square brackets show what to do if the condition isn't true: print the message "sorry, we only have chocolate."

When using an `ifElse` command, the second list of instructions is supposed to start on the same line as the ending of the first list. If I had done this, the left square bracket that starts the second list of instructions would have been on the same line as the right square bracket that ends the first list of instructions. (Don't confuse these with the square brackets that show the two `print` instructions what to print.) Because I couldn't fit both lists on one line, I used the tilde character (~) to say "treat the next line as if it were starting right here." You can use that anywhere you want in Logo; I just haven't used it yet because we've been using such short instructions.



```
ifElse :flavor = "chocolate [
  print [here's a chocolate cone]
] ~
[
  print [sorry, we only have chocolate]
]
```

Connecting the two lists with the tilde character

Note how each one-line "list" of instructions in this program (the single `print` instruction in each list) is indented two spaces from the square brackets that surround it. If either list had more than one instruction, indenting those instructions together would make it clearer that they work together as a group, which would make the program easier to read.

* **Save your `iceCream.logo` file, exit out of your editor, and run the `iceCream` program that you created. Run it several times, and answer "vanilla" or "chocolate" or "strawberry" or "grasshopper" or anything you like. Try running it several times and give it different answers, and see what it does. Try answering with a number or with a blank line. How does it respond?**

Starting Your Math Game

Now you know all you need to know to create the simplest version of your math game. It will ask one math question, wait for the answer, and then tell the user whether the answer was correct or not. (Later, we'll have it ask many math questions and keep score of how many correct answers were given.)

Let's look at the program before you type it in to see how it uses the various tricks you've learned.

```
; mathGame.logo
; (your name here) (today's date here)
; Give the user a math quiz.

to mathGame

  ; Pick the two numbers to add and save their total.
  make "addend1 random 10
  make "addend2 random 10
  make "total addend1+addend2

  ; Ask the user the question and get the answer.
  print (sentence "what "is :addend1 "+ :addend2 "?)
  make "answer readWord

  ; See if the user is right or wrong and let them know.
  ifElse  :total = :answer [
    print [You're right!]
  ]~
  [
    print (sentence "No, "it's :total)
  ]

end
```

The program has three sections, with a blank line between each section and a comment at the start of each section describing what it does:

1. The first section creates three variables. The first two variables, the two addends, are random numbers between 0 and 9. Because they're random numbers, the program will ask a different math question almost every time you run it. The third variable, `total`, is the sum of the two addends.

2. The second section prints the math question and reads the user's answer into the `answer` variable. To ask this math question, the program uses the `sentence` operation to combine the words in the question with the values assigned to the two addend variables. For example, if `addend1` has the value 7 and `addend2` is 3, the printed question will be "What is 7 + 3?"

3. The third section has an `ifElse` command. It compares the user's answer, which was stored in the `answer` variable in the second section, with the right answer, which was stored in the `total` variable in the first section. If it's true that these two are equal, the program prints "You're right!" If it's not true, it could just print "You're wrong," but it does better than that: it uses the `sentence` operation to combine the right answer (the `total` variable) with words to make a message telling the user the right answer. For example, if `total` has a value of 10, the message will be "No, it's 10".

- **Create a program file called `mathGame.logo` by entering the following instruction at the question mark prompt:**

```
editFile "mathGame.logo
```

- **Type in the lines shown above to create your `mathGame.logo` program.**

- **Save the program, exit it, and run your `mathGame` program a few times. If you always get the right answer, try entering the wrong answer on purpose—it's important to check that your program reacts the way you expect to everything your user might do. What if you answer the math question with a word instead of a number? What if you answer with a blank line?**

Try This!

If the two addends are always less than 10, then the math will be pretty easy. What would you change in your program to make the math problems harder? Try it.

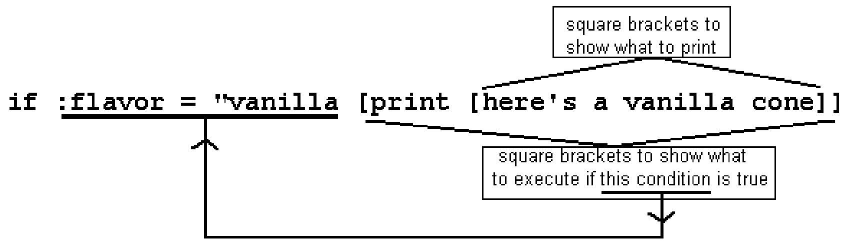
"While" Loops

Earlier in this chapter we saw how the `repeat` command tells Logo to execute the same list of instructions more than once. Do you remember how many times Logo executes the instructions in a list when you use the `repeat` command? As many times as you tell it to! For example, the following instruction gives Logo a list with two instructions in it (move forward 50 turtle steps and then turn right 90 degrees) and tells Logo to execute those two instructions four times:

```
repeat 4 [forward 50 right 90]
```

What if you want Logo to repeat a list multiple times, but you don't know yet how many times? For example, we want to set up our math game so that it keeps asking math questions as long as the user wants to answer them.

We can do this with a `while` loop. A `while` loop executes its instruction list as long as a certain condition is true. It's a lot like the `if` command that we saw before, which had a condition that was either true or false and a list of instructions to execute if the condition was true. For example, in the following `if` instruction, the condition that is either true or false is `:flavor = "vanilla` and the part in square brackets is what gets executed if the condition is true.



```
if :flavor = "vanilla [print [here's a vanilla cone]]
```

square brackets to show what to print
square brackets to show what to execute if this condition is true

Reviewing the if statement: which square brackets do what

For a Logo `while` loop, the condition goes inside of square brackets, just like the part to execute. Before we add this to our math game program, let's try `while` with a smaller program to get used to it.

* **Enter the following instruction to tell your text editor to create a new program file called `whileTest.logo`:**

```
editFile "whileTest.logo
```

* **Type in the following text to create a `whileTest` procedure in the `whileTest.logo` program:**

```
; whileTest.logo
; (your name here) (today's date here)
; Play with the while command.

to whileTest

  make "loopAgain "y   ; Initialize the loopAgain variable

  ; Keep asking user if Logo should loop again.
  while [:loopAgain = "y] [
    print [Should Logo loop again (y/n)?]
    make "loopAgain readWord
  ]

  ; Finishing message.
  print [I guess you're all done!]

end
```

In between the `to whileTest` line and the `end` line that start and end the procedure, `whileTest` has three parts:

1. **The first part creates a variable named `loopAgain` with a value of "y".**

2. **The second part has a list of two instructions (the `print` line and the `make` line) that it repeats while the condition `:loopAgain = "y` is true. Note how these two instructions are indented two spaces from the square braces that surround them, making it clearer that they work together as a group. You should always do this with the instructions inside of a `while` loop.**

   Look back at the first part of the program again, in its comment: what does it mean to *initialize* (pronounced "ih-nih-shull-ize") a variable? Well, imagine what would happen to the loop in this program's second part if the `loopAgain` variable hadn't been created first. The loop's instructions would never be executed, because when the loop's test condition got checked, and it asked the question "does `loopAgain` equal 'y'?" the answer would be "no" the first time the question was asked. The `loopAgain` variable can't equal "y" if it doesn't even exist.

   To initialize a variable is to give it a certain value to start with because a later part of the program will use that variable. Programs may act weird when they try to check variables that weren't properly created and initialized first, no matter what language they were written in. Because of this, certain languages such as Java won't even let you run your programs unless you first make sure that everything that might need to be initialized really is.

3. **The third part of the `whileTest` procedure, which happens after the `while` loop has executed its list of instructions for the last time, tells the user that he or she is all done with the program.**

* **Save your program, leave the editor, and run the `whileTest` program. Try answering `y` and then `n` once. Try answering with `y` many times before entering `n`. Try answering with `n` to the first question.**

Try This!

Run the `whileTest` program and answer with a letter that isn't `y` or `n`, like `x` or `z` or a number. What does the program do? Why?

Can you put any instructions you want inside of a `while` loop? Sure, but remember that the list of instructions to execute must include something that keeps the condition from being true forever. For example, the loop above runs as long as the `loopAgain` variable has the value "y", and the instruction

```
make "loopAgain readWord
```

allows `loopAgain` to be set to a value other than "y" so that the loop can finish. What do you do if a loop runs without ever stopping?

* **The following `while` loop is short enough that you don't need to create a program file for it. Just enter it at the question mark prompt:**

```
while [2+2 = 4] [print [yeah!]]
```

* **When you execute this instruction, it prints "yeah!" on the screen over and over and over. How long will it do this? Well, how long will two plus two equal four? Forever! Programmers call this "being stuck in a loop." Fortunately, UCB Logo does provide a way to break out of such a loop:**

  - **If you're using a computer running Windows, hold down the Ctrl key and type the letter Q.**
  - **If you're using a computer running Linux, hold down the Ctrl key and type the letter C.**
  - **If you're using a Macintosh, hold down the command key and type a period (.).**

If you're not using Logo and your computer ever "hangs," or gets stuck and ignores everything you do, it's probably stuck in a loop in one of the programs it's running. For example, the operating system program itself could be stuck in a loop. Sometimes there's just no way out of this, and you have to turn your computer off and start it up again. At least if Logo gets stuck in a loop, you now know how to break out of it, so remember the keystroke to do this on your computer!

Repeating the Math Questions and Keeping Score

Now that you know how to create a `while` loop, let's wrap the math game in one so that its users can play as many times as they want each time they start the program up.

* **Bring up the `mathGame.logo` file that you created so that you can edit it.**

```
editFile "mathGame.logo
```

* **Add the lines shown in bold below. The lines that are already there between the `to mathGame` line and the `end` line (along with the new lines that ask the user if he or she wants to play again) will be the body of your `while` loop. To indicate that these lines all work together as part of the `while` loop, indent them two spaces past the square braces that enclose them as shown below.**

```
; mathGame.logo
; (your name here) (today's date here)
; Give the user a math quiz.

to mathGame

  make "playAgain "y   ; Initialize the playAgain variable
  ; Keep asking if Logo should play again.
  while [:playAgain = "y] [

    ; Pick the two numbers to add and save their total.
    make "addend1 random 10
    make "addend2 random 10
    make "total addend1+addend2

    ; Ask the user the question and get the answer.
    print (sentence "what "is :addend1 "+ addend2 "?)
    make "answer readWord

    ; See if the user is right or wrong and let them know.
    ifElse  :total = :answer [
      print [You're right!]
    ]~
    [
      print (sentence "No, "it's :total)
    ]

    ; Ask the user if he or she wants to play again.
    print [Answer another math question (y/n)?]
    make "playAgain readWord
  ]

  ; When Logo gets here, the game is done.
  print [I hope you enjoyed mathGame!]

end
```

* **Save your file and try your game. Don't play it too many times yet, because we're about to make it even better by adding just a few more lines.**

Let's have `mathGame` keep track of how many questions were answered correctly and how many were answered incorrectly.

* **Edit the `mathGame.logo` file again and add the lines shown in bold type below.**

  The final `print` instruction, which tells the user what the score was, is long enough that I split it onto two lines. (Remember, the ~ tilde character tells Logo "look on the next line for the rest of this instruction.") You can leave out the tilde and write this all on one line if you want. If you do spread it out over two lines with the tilde, indent the second line as shown to make it clearer to people reading your program that it's not the beginning of a new instruction.

```
; mathGame.logo
; (your name here) (today's date here)
; Give the user a math quiz.

to mathGame
```

```
  make "rightAnswers 0 ; Initialize variables that keep track
  make "wrongAnswers 0 ; of right and wrong answers.

  make "playAgain "y   ; Initialize the playAgain variable

  ; Keep asking if Logo should play again.
  while [:playAgain = "y] [

    ; Pick the two numbers to add and save their total.
    make "addend1 random 10
    make "addend2 random 10
    make "total addend1+addend2

    ; Ask the user the question and get the answer.
    print (sentence "what "is :addend1 "+ addend2 "?)
    make "answer readWord

    ; See if the user is right or wrong and let them know.
    ifElse  :total = :answer [
      print [You're right!]
      make "rightAnswers :rightAnswers+1
    ]~
    [
      print (sentence "No, "it's :total)
      make "wrongAnswers :wrongAnswers+1
    ]

    ; Ask the user if he or she wants to play again.
    print [Answer another math question (y/n)?]
    make "playAgain readWord
  ]

  ; When Logo gets here, the game is done.
  print [I hope you enjoyed mathGame!]
  print (sentence "You "answered :rightAnswers ~
         "correctly "and :wrongAnswers "incorrectly.)

end
```

***	Save your revised program and run it. Throw in a few wrong answers on purpose to make sure that they're being counted correctly.**

Try This!

Think of a score and then try to make `mathGame` end up with that score at the end. Can you make it come out with zero right answers and zero

> wrong answers at the end? If not, why not? Look at the program's source code for hints.

You added three parts to this program to make it keep score. The first and third should be familiar:

1. The two new variables get initialized at 0.

2. Each time the user answers a math question, either the `rightAnswers` or `wrongAnswers` variable is adjusted to reflect the new score.

3. At the end, the program uses the `sentence` operation to output a message combining the `rightAnswers` and `wrongAnswers` variables with the words that make a complete sentence to print on the screen.

Let's look more closely at how one of the variables is adjusted:

```
make "rightAnswers :rightAnswers+1
```

Because the program initializes `rightAnswers` with a value of 0, the first time this instruction gets executed it's almost the same as saying this:

```
make "rightAnswers 0+1
```

Zero plus one is one, so `rightAnswers` then has the value 1 instead of 0. The next time this gets executed, it will be like saying this,

```
make "rightAnswers 1+1
```

so `rightAnswers` will equal 2.

To take a variable with a number in it and then add one to that number is something that computer programs do all the time, because programs often need to track how many times something happened. Adding one to a number stored in a variable happens so often that we have a special word for it: ***incrementing*** the variable's value. Subtracting one from the value of a variable also happens often, and we call that ***decrementing*** the variable's value.

Your `mathGame` program keeps track of how many times two different things happen: how many times right answers are entered and how many times wrong answers are entered. That's why it has two variables to keep track of these.

Improving the Program's Interface

Earlier in this chapter, we found out that an interactive program is one that the user can interact with. The user can change what the program does by clicking the mouse, typing in instructions, picking things off of menus, or performing other actions to tell the program what to do. The ***interface*** of a program is the parts that a user sees and deals with and the steps that the user takes to get anything done. For example, the interface of an e-mail program might consist of the menus and icons that you click on to tell it what to do. The interface of your `mathGame` program is the text messages that appear on the screen to give the user clues about what it expects, as well as the program's pauses to wait for the user's input.

Try This!

Play `mathGame`, but type in the word "hello" when it asks you for the sum of the two addends and enter it again when `mathGame` asks you if you want to answer another math question. How does it react? Try again, but this time don't answer anything in response to these questions—just press the **Enter** key. What does it do this time? Look over the program's code again. Why do you think it did what it did? How a program responds to bad input is part of its interface as much as how it responds to good input.

How do you improve an interface? A good start is to look for things about running the program that are annoying and then try to make them less annoying. For example, when running `mathGame`, I think it's annoying that I must type a "y" and press **Enter** after every single math question. If I want to answer 15 math questions, I don't like pressing "y" 14 times after the first math question. My answer will usually be "yes," so the program would be easier to use if, when it asks me about answering another math question, pressing **Enter** without typing anything gets treated the same as entering "y" and then pressing **Enter**.

Another way to improve the interface is with ***error checking***. This consists of figuring out problems that might happen and adding instructions to the program to make it easier for the user to see and correct these problems. For example, if `mathGame` asks the user what 2+3 is and the user responds with "hello," it would be nice if `mathGame` could tell the user " 'hello' is not a number. Try again."

We will improve the interface of `mathGame` by making both of these changes. The first change is simple: if the user responds to the prompt about playing again by just pressing **Enter**, we don't want the `while` loop to end, so even though the `playAgain` variable won't have a "y" in it, we'll put one in it to make sure that the `while` loop keeps looping.

* **Bring up the `mathGame.logo` file that you created so that you can edit it.**

```
editFile "mathGame.logo
```

* **In your program, find the first three lines of the text shown below. Put square brackets around the "y" at the end of the second line. This shows the user that "y" is the default choice. (In other words, if they don't pick a choice, then that's the one that the program will assume that they want.) Although we've seen square brackets used several times in Logo programs before, this pair of them has nothing to do with Logo. It won't affect how the program runs; it's there to show the user something—it's part of the user interface!**

```
         ; Ask the user if he or she wants to play again.
         print [Answer another math question ([y]/n)?]
         make "playAgain readWord

         ; If the user just pressed Enter, reset playAgain to "y"
         if empty? :playAgain [
                make "playAgain "y
         ]

  ]

  ; When Logo gets here, the game is done.
  print [I hope you enjoyed mathGame!]
  print (sentence "You "answered :rightAnswers ~
         "correctly "and :wrongAnswers "incorrectly.)
end
```

* **Under the line with the ([y]/n) part, skip a line and add the bolded four lines shown above (the comment and the three-line if instruction). The empty? part is an example of a special part of Logo called *predicates* (pronounced "pred-ih-kits") which are often spelled with a question mark at the end. Logo treats this particular predicate, along with the part after it (:playAgain) as conditions that are either true or false, depending on whether playAgain has a value in it or is empty. Logo treats all predicates, along with any input information that goes with them, as a condition that's either true or false, so predicates are often used with if commands and while loops. Think of the predicate as asking a short question about the input right after it. Above, it's asking "is playAgain empty?" Other predicates ask other questions; we'll meet another shortly.**

  If playAgain is empty, the if instruction will execute the instruction between the square braces, which sets playAgain back to "y." If playAgain is not empty, the if instruction will skip past the part in the square braces, leaving the playAgain variable alone, and the program will continue the same way it did before you added these lines.

* **Save your changes and run mathGame again. Try answering the "Play again ([y]/n)?" question with different answers: y, n, a carriage return with no other characters, a number, a different letter, and anything else you can think of. Does mathGame behave the way you thought it would?**

Another handy predicate is number?. (As with empty?, the question mark is part of how it's spelled). Logo treats it and the input after it as a true condition if the input is a number and as a false condition if the input isn't. Let's play with it a little before we put it to work for us.

* **First, let's see what Logo does when we tell it to print a condition that is true or false. Enter and execute the following instruction at the question mark prompt:**

```
print 3 = 3
```

  This is like saying "tell me whether this is true or not."

* **Try it again when something that isn't true:**

```
print 3 = 4
```

Now you've got the idea: if you tell Logo to print a condition is either true or false, it prints the word "true" or the word "false."

* **Let's try this with the `number?` predicate. Execute the following instruction:**

```
print number? 3
```

This is like asking Logo "Is it true or false that 3 is a number? Print the answer."

* **Try something that you know isn't a number:**

```
print number? "hello
```

* **A predicate's input can also be a value stored in a variable. Enter these two instructions:**

```
make "testVar 5
print number? :testVar
```

* **Try this again with something that isn't a number stored in `testVar`:**

```
make "testVar "dog
print number? :testVar
```

* **What if you have Logo do some math and output the answer to the `number?` predicate? Try this:**

```
print number? 3+2
```

Earlier we said that it would be nice, when `mathGame` asks the user what 2+3 is and the user responds with "hello," if `mathGame` could tell the user " 'hello' is not a number. Try again." Now we have what we need to do this, because the `number?` predicate can check whether the user's response is a number. How many times should we ask? As many as it takes to get the user to answer with a number. How do we tell Logo to ask over and over until the user enters a proper number? With a `while` loop! Let's try this with a little program as a test before we add it to the `mathGame` program.

* **Create a new program file called `numberTest.logo`.**

```
editFile "numberTest.logo
```

* **Type in the following text for your program. (Because it's just a little test program, I got lazy and didn't add any comments.)**

```
to numberTest

  make "userAnswer "hello
  while [not number? :userAnswer] [
    print [Enter a number.]
    make "userAnswer readWord
    if not number? :userAnswer [
      print [That's not a number.]
    ]
  ]

  print [OK! That's a number!]

end
```

We saw before that if a `while` loop's test condition checks a variable's value, then that variable should be initialized first. It's a little trickier here than when we did it with the `playAgain` variable. We want the loop to keep repeating as long as the variable *doesn't* have a number in it, so there are two things we have to remember:

- **Once `number? :userAnswer` turns out to be true, we're all done with the loop, because we'll have a number in `userAnswer`. We want to keep going when it *isn't* true, so we use the `not` operation in the condition at the beginning of the loop. This `while` loop is saying "While the value in `userAnswer` is not a number, keep executing the instructions inside the square brackets." The `if` instruction also uses the `not` operation to check whether it should print the message "That's not a number."**
- **If `userAnswer` is initialized with a number, the `while` loop's list of instructions won't even execute once, and the user won't get to enter anything, so it must be initialized with something that isn't a number. I picked the word "hello," but could have picked anything.**

* **Save the program, quit the text editor, and run your `numberTest` program a few times.**

Try This!

A few pages ago you used the `print` command to ask Logo whether various conditions (`3 = 3`, `3 = 4`, `number? 3`, and more) were true or false. Try them again, but put the operation `not` after the `print` command like this: `print not 3 = 3`. This is like telling Logo "print whether the following condition is *not* true." Or, it's like saying "if the following condition is true, print 'false,' and if the condition is false, print 'true.' "

Using `not` to make things backward may seem like a confusing little puz-

> zle, but look how valuable it was in our `numberTest` program. Logo executes any `while` command's instruction list as long as the `while` condition is true. In `numberTest`, we wanted it to repeat as long as a certain condition ("Does `userAnswer` have a number in it?") was not true, because we wanted the program to keep asking the user to enter something until he or she entered a number. So, we put a `not` operation in front of the `number? :userAnswer` test so that as long as the test was false, the `while` command would see the whole condition (`not number? :userAnswer`) as true and keep repeating the `while` command's instruction list.

Now that we've got a better feel for how to use the `number?` predicate, we're ready to put it to work in the `mathGame` program to improve the program's user interface.

- * **Bring up the `mathGame.logo` program file in your text editor.**

```
editFile "mathGame.logo
```

- * **The text below that is not in bold text should already be in your `mathGame.logo` file. Find it and add the bolded text where it's shown here. Pay extra attention to the square brackets that you need to add, because if you miss one the program won't work.**

```
; Pick the two numbers to add and save their total.
make "addend1 random 10
make "addend2 random 10
make "total addend1+addend2

; Ask the user the question and get the answer.
make "answer "NotANumber
while [not number? :answer] [

  print (sentence "what "is :addend1 "+ addend2 "?)

  ; Check whether the answer is really a number.
  make "answer readWord
  if not number? :answer [
    print [Please answer with a number.]
  ]
]

; Once we reach this point, we know that :answer
; stores a number.

; See if the user is right or wrong and let them know.
ifElse  :total = :answer [
  print [You're right!]
  make "rightAnswers :rightAnswers+1
]~
[
  print (sentence "No, "it's :total)
  make "wrongAnswers :wrongAnswers+1
]
```

You won't completely leave the non-bold text that was already there alone. Now that some of it is inside of a loop, you should indent it to make the program easier to read.

Note how instead of saying "That's not a number" like our little `numberTest` program did, the error message in `mathGame` says "Please answer with a number." Being polite to the user is an important part of a good interface—no one likes an obnoxious program!

* **Save the program, quit the text editor, and run your `mathGame` program a few times. Have someone else try it. Ask them what they think of the interface, and if there's anything that could be improved about it.**

What We Learned

In this chapter, we learned:

- How to make Logo do math.
- How to use variables to do math.
- What interactive programs are and how to create a simple one.
- What the `random` operation does, and how to use it make strange pictures.
- How to have your program make decisions using the `if` and `ifElse` commands.
- How to have Logo execute a list of instructions over and over as long as a certain condition is true (or, if necessary, as long as it's not true) using a `while` loop.
- How to use the `empty?` and `number?` predicates to check on conditions.
- Why a program's interface is important, and a few ways to improve it.

New Commands and Operations in This Chapter

(See the "Procedures Reference" Appendix in the back of the book for brief descriptions of what these do.)

New Commands

- `clearText`
- `if`
- `ifElse`
- `while`

New Operations

- `empty?`
- `not`
- `number?`
- `random`
- `readWord`
- `repCount`
- `sentence`

More Things to Try

- Convert `mathGame` to ask the user multiplication facts instead of addition. Remember, Logo uses the asterisk (`*`) instead of the plus sign for multiplication.
- Convert `mathGame` to ask the user subtraction facts. This will be a little trickier, so here's a hint about the tricky part: after executing the following two instructions, what's the biggest value that `addend1` can have? Will it ever be bigger than `sum`?

  ```
  make "sum random 30
  make "addend1 random :sum
  ```
- Change `mathGame` to always ask 10 math questions instead of letting the user control how many questions it asks.

- Think of a program that you use often and list three things that could be done to improve its interface.
- Write an interactive graphical program. All it has to do is to ask the user for information and uses that information to affect how the turtle draws a picture.
- Look at the source code of the `ttt` program again. Are there any commands or operations that you didn't recognize before but that you recognize now? Is there anything that gives you ideas for things that you can add to the `mathGame` program?

Glossary

Highlighted terms in each definition have their own entries in the glossary.

binary ("bye-nurry") A binary file is usually arranged in a special way so that only certain programs that know about that arrangement know how to read that file off of a computer's disk and do something with it. When viewed with a ***text editor***, binary files look like a mess. Pictures and music are almost always stored in binary files. Compare this with ***text files***.

case-sensitive A case-sensitive program cares about the difference between upper- and lower-case letters. If it asks you to type "YES" and you type "yes," that won't be good enough. Logo is case-insensitive, so it considers "YES" and "yes" to be the same thing.

character A single letter, numeric digit, punctuation mark, or space. The word "dog" and the number "100" each have three characters in them.

code A short version of the term "source code." Programmers sometimes use these terms to refer to the contents of a program that they typed. Typing instructions into a program is often called "coding." If a program has to be compiled, or converted into something else in order for a computer to run it, "source code" refers to the original instructions written for the computer before they were compiled.

command In Logo, a special word, or procedure, that tells Logo to do something. (Outside of Logo, the word "command" often refers to what Logo calls an ***instruction***. Compare the definition of ***operation***.

command prompt One or more characters that appear on your screen to show you where to enter ***instructions***. In Logo, the command prompt is usually a question mark.

computer science The study of ***programming languages***, ***operating systems***, and other parts of building and using computer programs.

cursor A ***character*** on your screen, usually an underscore (_), that shows where your letters, numbers, and spaces will appear on the screen when you type them on the computer keyboard.

declaring variables Listing out, at the beginning of a program, which ***variables*** the program uses. Some ***programming languages*** make you do this, but Logo doesn't.

decrement To subtract one from a number. If you decrement 5, you'll have 4; if you decrement 100, you'll have 99. Sometimes programs decrement number values stored in ***variables*** when they're counting something.

double quote A term that programmers often use for the quotation mark character (**"**). It's on the same key as the ***single quote***, but you need to press down the **Shift** key to type the double quote.

error checking Having your program check for things that might go wrong and then either correcting them or letting the ***user*** know about them.

error message A message to the ***user*** about something that went wrong. A good error message should be easy for the user to understand.

execute To execute a program or instruction is to run it. Once you type a Logo ***instruction*** at Logo's ***command***

prompt, press the **Enter** key (or, on a Macintosh, the **return** key) to execute that instruction.

grammar The rules for putting together the words of a language, whether it's a spoken language like English or Spanish or a ***programming language*** like Java or Logo.

increment To add one to a number. If you increment 5, you'll have 6; if you increment 100, you'll have 101. Sometimes programs increment number values stored in ***variables*** when they're counting something.

indent To put extra space at the beginning of a line to move it over.

```
; This Logo comment is not indented.
  ; This Logo comment is indented two spaces.
    ; This Logo comment is indented four spaces.
    ; This one is also indented four spaces.
```

Programmers often indent a group of instructions (for example, the instructions between the square brackets of an `if` or `while` instruction) the same amount to show that they work together.

initialize ("ih-nih-shull-ize") If part of your program checks the value of a ***variable***, but that variable doesn't have a value, the program could have problems. Putting a value into a variable to make sure that it has a value is called initializing it.

input Information coming into a computer. Input might come from a file on a disk, or from another computer hooked up to yours by a phone line, or from a microphone that you're singing into, or from a keyboard that you're typing on. Logo has a special meaning for "input"; see the definition for ***parameters***. (Also, compare the definition of ***output***.)

instruction A combination of one or more Logo ***commands*** and ***operations*** and the ***input*** that they need to do their job. `forward` is a command that needs a number after it; `forward 50` is a complete instruction.

Usually, a complete line that you type at the Logo question mark ***prompt*** or in a Logo ***program*** (like `forward 50`) is an instruction, although an instruction can be spread out over multiple lines. You can also put multiple instructions on one line.

interactive An interactive program is one that the ***user*** can interact with. In other words, the user can change what the program does by clicking on the mouse, typing in instructions, picking things off of menus, or even by talking into a microphone or turning a steering wheel attached to the computer. Almost every program you've ever used on a computer is interactive.

interface The parts of a program that a ***user*** sees and deals with, and the steps that the user must take to get anything done. For example, the interface of an e-mail program might consist of the menus and icons that you click on to tell it what you want it to do. The interface of your `mathGame` program that you create in this book is the text scrolling up the screen giving you clues about what it expects of you and the pauses that wait for your input.

Linux ("linnix") An ***operating system*** that runs on computers of all sizes. Linux is free and very popular with programmers.

Logo A ***programming language*** invented over thirty years ago to make it easier for kids to learn programming.

Logo interpreter A program that understands the Logo ***programming language***. It can execute instructions one at a time as you enter them at the ***command prompt***, and it can also execute lists of instructions saved as ***programs***. The pretend "turtle" used to draw graphics in Logo makes it easy to have fun with this programming language without knowing too many different commands.

loop A list of instructions that get repeated in a program. They may get repeated for a set number of times or they may get repeated as long as while a certain condition is true.

monitor The glowing screen attached to your computer that shows what's going on. It looks like a television set.

operating system A special program that starts up as soon as you turn on a computer. You can't do anything else with the computer until the operating system is up and running. When you click an icon on your screen or pick something on a menu to start up a program, you're actually telling the operating program to start that program up. When you send a picture or a story to the printer, you're really telling the operating system to send it to the printer.

operation In Logo, a special word, or procedure, that outputs a value to be used by something else—usually by a command, but sometimes by another operation. (If it's used by another operation, there has to be a command somewhere in the instruction using the result of all this outputting.) Compare the definition of ***command***.

output Information coming out of a computer. It might come out onto your monitor screen, or onto your printer, or out of a speaker attached to your computer, or out over a phone line hooked up to another computer. Compare the definition of ***input***.

parameters ("puh-ram-ih-terz") Pieces of information that certain ***procedures*** need to do their job, like the number that you put after a `forward` command. Logo calls these ***inputs***, but other programming languages call them parameters.

predicate ("pred-ih-kit") A Logo ***operation*** that lets you figure out whether a certain condition is true or false. They're very handy in `if` statements and `while` ***loops***.

procedure ("pro-seed-jer") Nearly all of the special words that mean something in the Logo ***programming language*** are procedure names. There are two kinds of procedures: ***commands*** and ***operations***. Logo comes with its own built-in procedures (for example, commands such as `forward` and `make` and operations such as `readWord` and `random`) and you can write your own new ones.

program A list of ***instructions*** that are saved together and named. That name becomes a new ***procedure*** in Logo, and someone can ***execute*** that procedure just like they can execute the procedures that are already a part of Logo.

programming language A language for giving instructions to a computer. Each programming language, like each spoken language, has its own vocabulary (the list of words that it knows) and grammar (the rules for putting those words together). Different programming languages are good at different things; Logo is famous for being a good one for kids to start with because it's so much fun.

prompt See ***command prompt***.

scientific notation A special way of writing numbers that scientists and mathematicians use to write very big or very small numbers.

scrolling Usually, this describes what happens when all the text on a screen moves up on the screen, just as if it was printed on a long scroll of paper and someone rolled up the top of the scroll a little and unrolled the bottom of the scroll a little.

single quote A term that programmers often use for the apostrophe character (`'`). It's on the same key as the ***double quote***.

source code See ***code***.

string A word, a phrase, or any "string" of characters that is just supposed to be characters and not a number to do math with or some other special type of data. "hello" and your phone number and even a group of spaces ("   ") are all strings.

text editor A program that lets you edit ***text files***. Many text editors have special features that make it easier to edit program ***source code***.

text files A file that's not a ***binary file***. A text file is usualy made up of letters, numbers, and symbols that you type on your keyboard. Text files can be read by all kinds of programs on all kinds of computers. Unlike most binary files, a text file from one computer can easily be moved to another computer and used there, even if the computer is running a different ***operating system***. Computer program ***source code*** files are always text files.

user The person using a computer program. When programmers talk about users, they're talking about the people they're creating the program for.

variables ("vair-ee-uh-bulls") Containers for information, usually in a program. Programs can pass variables around from one command to another, they can look inside them to see what information they store, and they can change the information inside. It's very common for computer programs to spend much of their time looking inside of variables, checking the information, and then doing different things depending on what they find there.