1 Exploration

The name Logo comes from the Greek word *logos,* which means "word." In contrast to earlier programming languages, which emphasized arithmetic computation, Logo was designed to manipulate language—words and sentences.

Like any programming language, Logo is a general-purpose tool that can be approached in many ways. Logo programming can be understood at different levels of sophistication. It has been taught to four-year-olds and to college students. Most of the books about Logo so far have been introductory books for young beginners, but *this* book is different. It's for somewhat older learners, probably with some prior computer experience, although not necessarily Logo experience.

This book was written using the Berkeley Logo dialect, a version of Logo that's available at no cost for PCs, Macintoshes, and Unix systems. Recent commercial Logo dialects have emphasized the control of real-time animation, robotics, and other such application areas, somewhat at the expense of more traditional Logo features designed to be useful in the development of larger and more complex programs. Berkeley Logo follows the traditional design, so you may miss some "bells and whistles" that you associate with Logo from elementary school. In fact, we'll hardly do any graphics in this book!

Some of the details you'll have to know in order to work with Logo depend on the particular kind of computer you're using. This book assumes you already know some things about your computer:

- How to turn on your computer and start Logo
- How to type a command, ending with the RETURN key
- How to use control keys to correct typing mistakes
- How to use a text editing program

These points I've listed aren't actually part of the Logo *language* itself, but they're part of the Logo programming *environment*. Appendix A has a brief guide to some of these machine-specific aspects, but if you've never used a computer before at all, start by working with some application programs to get the feel of the machine.

On the other hand, I'd like to pretend that you know nothing about the Logo language—the primitive procedures, the process of procedure definition, and so on—even if you've really used Logo in elementary school. The reason for this pretense is that I want you to think about programming in what will probably be a new way. The *programs* may not be new to you, but the *vocabulary* with which you think about them will be. I'm warning you about this ahead of time because I don't want you to skip over the early chapters, thinking that you already know what's in them.

Okay, it's time to start Logo running on your computer. You should then see a screen that says something like

```
Welcome to Berkeley Logo version 3.3
?
```

The question mark is Logo's *prompt*. When you see the question mark, it means that the computer is prepared for you to type in a Logo *instruction* and that Logo will carry out the instruction as soon as you finish it.

Getting Acquainted with Logo...

Right now, type this instruction:

```
repeat 50 [setcursor list random 75 random 20 type "Hi]
```

Remember that square brackets `[]` are different from parentheses `()`. Also remember that it's important to put spaces between words. However, it doesn't matter whether you use `UPPER CASE` or `lower case` letters in the words that Logo understands.

If all goes well, Logo will cheerfully greet you by scattering `Hi`s all over the screen. If all doesn't go well, you probably misspelled something. Take a look at what you typed, and try again.

Afterward, you can clear the screen by typing `cleartext` or its abbreviation `ct`.

... in Two Senses

I thought it would be appropriate to start exploring Logo by having it say hello. You and Logo can get acquainted as you would with another person.

But, of course, the point of the exercise is to get acquainted with Logo in a more serious sense too. You're seeing what a Logo instruction looks like and a little bit about what kinds of things Logo can do. In this first chapter the kind of acquaintance I have in mind is relatively superficial. I'm trying to get across a broad sense of Logo's flavor rather than a lot of details. So I'm not explaining completely what we're doing here. For that reason, the second chapter will repeat some of the same activities, but I'll give a more detailed discussion there.

Perhaps you've made Logo's acquaintance before, probably through the medium of turtle graphics. In that first introduction you may have explored Logo's ability to manipulate text as well as graphics. But maybe not. Writing a book like this, it's not easy for me to carry on a conversation with someone I haven't met, so in this introduction I may be saying too much or too little for your individual situation. I hope that by the second chapter you and the other readers will all be ready for the same discussion.

If you haven't used Logo before, or if you've used only the part of Logo that has to do with turtles, look at the instruction I asked you to type earlier. Think about the different parts of that instruction, the words like `repeat` and `random` and `setcursor`. Try to figure out what each one means. Then see if you can figure out an experiment to decide if you've understood each word correctly! Later, we'll go over all these details and you'll learn the "official" explanations. But the kind of experimenting I'm suggesting isn't pointless. This kind of exploration may raise questions in your mind, not just about the meanings of the Logo words but about how they're connected together in an instruction, or about *why* a word means just what it does rather than something a little different.

Another Greeting

Here is a somewhat less "scatterbrained" greeting instruction:

```
repeat 20 [repeat random 30 [type "Hi] print []]
```

Try that one. Compare it to the one we started with. Which do you like better? Do you prefer random scattering, or orderly rows? Perhaps this question will teach you something about your own personality!

Fooling Around

Then again, maybe you think this is all silly. If so, I'd like to try to convince you that there are some good, serious reasons for you to take a lighthearted approach to computer programming, no matter how serious your ultimate goals may be.

There are two aspects to learning how to program in a language like Logo. One aspect is memorizing the vocabulary, just as in learning to speak French. If you flip through the reference manual that came with your Logo,* you'll find that it's a sort of dictionary, translating each Logo word into a bunch of English words that explain it. But the second aspect is to learn the "feel" of Logo. What kinds of problems does Logo handle particularly well? What are the examples of programming *style* that correspond to the idioms of a human language? What do you do when something doesn't work?

It is by fooling around with Logo that you learn this second aspect of the language. Starting with the second chapter of this book, we'll be going through plenty of dry, carefully analyzed fine points of Logo usage. But as we progress, you should still be fooling around, on the computer, with the ideas in the chapters.

In fact, I think that that kind of intellectual play is the best reason for learning about computer programming in the first place. This is true whether you are a kid programming for the fun of it or an adult looking for a career change. The most successful computer programmers aren't the ones who approach programming as a task they have to carry out in order to get their paychecks. They're the ones for whom programming is a joyful game. Just as a baseball diamond is a good medium in which you can exercise your body, the computer is a good medium in which you can exercise your mind. That's the real virtue of the computer in education, not anything about job training or about arithmetic drill.

A Slightly Longer Conversation

The Logo words such as `print` and `random` are the names of *procedures,* little pieces of computer program that are "specialists" in some particular task. We are now going to add to Logo's repertoire by inventing a new procedure named `hi`. At the question mark prompt, start by typing this:

* If you're using Berkeley Logo, it's in a file named `usermanual` (or `userman.ual` if you're using a DOS machine) that should be installed along with the Logo program. The Berkeley Logo reference manual is also an appendix to Volume 2 of this series.

```
to hi
```

The word `to` here is short for "here's how to." The name is intended to suggest the *metaphor* that what you're doing when you write computer programs is to *teach* the computer a new skill. Metaphors like this can be very helpful to you in understanding a new idea. (Just ask any English teacher.) I'll point out other metaphors from time to time.

Logo should have responded to this instruction by printing a different prompt character. Instead of the question mark, you should now see a greater-than sign (`>`) at the beginning of the line:

```
? to hi
>
```

(Whenever I show an interaction with the computer in this book, I'll show the part that you're supposed to type in **`boldface`**; what the computer prints in response is in `lightface`. But I won't use boldface when I'm only showing what you type and not a complete interaction.) This new prompt means that Logo will not immediately carry out whatever instructions you type; instead Logo will remember these instructions as part of the new procedure `hi`. Continue typing these lines:

```
print [Hi. What's your name?]
print sentence [How are you,] word first readlist "?
ignore readlist
print [That's nice.]
end
```

Again, be careful about the spaces and punctuation. After the last line, the one that just says `end`, Logo should go back to the question mark prompt. Now just type

```
hi
```

on a line by itself. You can carry on a short conversation with this program. Here's what happened when I tried it.

```
? hi
Hi. What's your name?
Brian Harvey
How are you, Brian?
I'm fine.
That's nice.
```

If something unexpected happens when you try it, perhaps you made a typing mistake. If you know how, you can fix such mistakes using the Logo editor. If not, you'll have a chance to review that process later, but for now, just start over again but give the procedure a different name. For example, you can say

```
to hi2
```

for the second version of `hi`.

☞ This program pretends to be pretty smart. It carries on a conversation with you in English. But of course it isn't really smart. If you say "I feel terrible" instead of "I'm fine," the procedure cheerfully replies "That's nice" anyway. How else can you mess up the program? What programming tools would you need to be able to overcome the "bugs" in this program?

(When a paragraph starts with this symbol ☞ it means that the paragraph asks you to invent something. Often it will be a Logo program, but sometimes, as in this case, just answers to questions. This is a good opportunity to take a break from reading, and check on your understanding of what you've read.)

A Sneaky Greeting

This chapter started as a sort of pun in my mind—the one about getting acquainted. How should I have Logo introduce itself? I'm still playing with that idea. Here's another version.

```
to start
cleartext
print [Welcome to Berkeley Logo version 3.3]
type "|? |
process readlist
type "|? |
wait 100
print [Ha, ha, fooled you!!]
end

to process :instruction
test emptyp :instruction
iftrue [type "|? | process readlist stop]
iffalse [print sentence [|I don't know how  to|] first :instruction]
end
```

The vertical bars are used to tell Logo that you want to include space characters within a word. (Ordinarily Logo pays no attention to extra spaces between words.) This is the sort of grubby detail you may not want to bother with right now, but if you are a practical joker you may find it worth the effort.

A Quiz Program

Before we get on to the next chapter, I'll just show you one more little program. Try typing this in. As before, you'll see greater-than prompts instead of question marks while you're doing it.

```
to music.quiz
print [Who is the greatest musician of all time?]
if equalp readlist [John Lennon] [print [That's right!] stop]
print [No, silly, it's John Lennon.]
end
```

You can try out this procedure by typing its name as an instruction.*

☞ If you don't like my question, you could make up your own procedures that ask different questions. Let's say you make up one called `sports.quiz` and another called `history.quiz`, each asking and answering one question. You could then put them all together into one big quiz like this:

```
to total.quiz
music.quiz
sports.quiz
history.quiz
end
```

Saving Your Work

If you do write a collection of quiz procedures, you'll want to save them so that they'll still be available the next time you use Logo. Certainly you'll want to save the work you

* It has been suggested by some reviewers of the manuscript that there may be younger readers who don't know who John Lennon is. Well, he's the father of Julian Lennon, an obscure rock star of the '80s, and he used to be in a rock group called the Quarrymen. If you have trouble with some of the cultural references later in the book you'll have to research them yourself.

do in later chapters. You can ask Logo to record all of the definitions you've made as a *workspace* file using the `save` command. For example, if you enter the instruction

```
save "mystuff
```

you are asking Logo to write a disk file called `mystuff` containing everything you've defined. (The next time you use Logo, you can get back your definitions with the `load` command.)

Don't get confused about the difference between a *procedure* name and a *workspace* name. Logo beginners sometimes think that `save` saves only a single procedure, the one whose name you tell it (in this example, a procedure named `mystuff`). But the workspace file named `mystuff` will actually contain *all* the procedures you've defined. In fact, you probably don't have a procedure named `mystuff`.

The format for the name of a disk file will depend on the kind of computer you're using, whether you're writing to a hard disk or a floppy disk, and so on. Just use whatever file name format your system requires in other programs, preceded by the quotation mark that tells Logo you're providing a word as the input to the `save` command.

About Chapter 2

In this chapter the emphasis has been on *doing* things. You've been playing around with some fairly intricate Logo instructions, and if you don't understand everything about the examples, don't let that worry you.

Chapter 2 has the opposite emphasis. There is very little to do, and the examples will seem quite simple, perhaps even insultingly simple! But the focus of the chapter is on *understanding* those simple examples in great detail.

Logo deserves its reputation as an easy-to-learn language, but it is also a very sophisticated one. The ease with which Logo can be learned has lured many people into sloppy thinking habits that make it hard for them to grow beyond the most trivial programming. By studying examples that seem easy on the surface, we can start exploring *below* the surface. The important questions will not be ones like "what does `print` do," but instead ones like "what is going on *inside* the Logo interpreter when I type `print`?"

Later chapters will strike more of a balance between things to do and things to think about. If the pace seems slow in chapter 2, glance back at the table of contents to reassure yourself about how much territory we'll cover before the end of the book. Then keep in mind that you'll need the ideas from chapter 2 in order to understand what comes later.

No Exercises

This is the point in the chapter where you might be expecting a set of exercises: Problem 1.1, get the computer to print your name.

There aren't any exercises—but not because you shouldn't try using Logo at this point. The reason is that part of the challenge is for *you* to invent things to try, not just rely on me for your ideas. In each chapter there will be some sample procedures to illustrate the new information in the chapter. You should try to invent programs that use those ideas.

But I hope it's clear by now that I don't want you to do this with a sense of duty. You should play with the ideas in each chapter only to the extent that it's interesting and mind-stretching for you to do so.

In this chapter I really haven't yet told you any of the rules for putting together Logo instructions. (I'll do that in Chapter 2.) So you shouldn't get discouraged or feel stupid if you don't get very far, right now, in playing with Logo. It will be a few more chapters before you should expect to feel really *confident* about undertaking new projects of your own. But you won't break anything by trying now. Go ahead, fool around!